robotron

SOFTWARE
DOKUMENTATION

Arithmetikprozessor

Anleitung
fuer
den Programmierer

- Arithmetikprozessor -

VEB Robotron Buchungsmaschinenwerk
Karl-Marx-Stadt
VEB Robotron Bueromaschinenwerk
Soemmerda

*** ARITHMRTIKPROZESSOR ***

Die vorliegende 1. Auflage der Dokumentation "Arithmetikprozessor" fuer PC EC 1834 entspricht dem Stand vom 31.12.87 und unterliegt nicht dem Aenderungsdienst.

Nachdruck, jegliche Vervielfaeltigung oder Auszuege daraus sind unzulaessig.

Die Dokumentation wurde durch ein Kollektiv des

VEB Robotron Buchungsmaschinenwerk Karl-Marx-Stadt

erarbeitet.

Bitte senden Sie uns Ihre Hinweise, Kritiken, Wuensche oder Forderungen zur Dokumentation zu.

Die Dokumentation "Arithmetikprozessor" ist eine Ergaenzung zur "Anleitung fuer den Assemblerprogrammierer".

Diese Beschreibung ist im Zusammenhang mit dem Teil I der "Anleitung fuer den Assemblerprogrammierer zu verwenden.

Sie beschreibt das Zusammenspiel Hauptprozessor und Arithmetikprozessor sowie die Befehle des Arithmetikprozessors.

VEB Robotron Buchungsmaschinenwerk
Karl-Marx-Stadt
PSF 129
Karl-Marx-Stadt
7010

*** ARITHMRTIKPROZESSOR ***

I N H A L T S V E R Z E I C H N I S

		Seite
1.	Moeglichkeiten des Arithmetikprozessors	4
2.	Mikroprozessorsystem WM86 / WM87	6
2.1.	Befehlsausfuehrung	6
2.2.	Synchronisation der Befehle	7
3.	Architektur des WM87	8
3.1.	Gleitkommastack	8
3.2.	Stackregister adressieren	9
4.	Steuerungsumgebung	9
4.1.	Statuswort	10
4.2.	Ausnahmen (Exceptions)	11
4.3.	Behandlung der Ausnahmen	12
4.4.	Kontrollwort	13
4.5.	Identifizierungswort (Tag Word)	15
4.6.	Exception-Pointers	15
4.7.	Interruptverarbeitung	16
4.8.	Exception Handler	16
5.	Datentypen und Datenformate	17
5.1.	Binary Integer	18
5.2.	Packed Decimal Integer	18
5.3.	Real	19
5.4.	Spezielle Real-Zahlen	19
5.5.	Zahlenbereich von Reals	24
6.	Datendefinition, Adressierung	27
6.1.	Definieren von Daten	27
6.2.	Adressieren der WM87-Operanden	28
7.	Befehle des Arithmetikprozessors	30
7.1.	Datentransferbefehle	30
7.2.	Arithmetische Befehle	39
7.3.	Vergleichsbefehle	63
7.4.	Transzendentale Befehle	72
7.5.	Konstanten	76
7.6.	Prozessorsteuerbefehle	80
8.	Befehlsuebersicht	95
Anlage 1:	Ausnahmen und ihre Antworten	98
Anlage 2:	Definitionen / Abkuerzungen	105

DER__ARITHMETIKPROZESSOR__K1810_WM87

1._Moeglichkeiten_des_Arithmetikprozessors

Der Arithmetikprozessor K1810 WM87 (in dieser Beschreibung als WM87 bezeichnet) gehoert zur Familie des K1810 WM86 (nachfolgend als WM86 bezeichnet) und kann mit dem 16-Bit-Mikroprozessor WM86 zusammenarbeiten.

Er verarbeitet parallel zum Hauptprozessor sieben verschiedene Datentypen und ca. 120 zusaetzliche numerische Befehle. Diese Befehle sind im Befehlsstrom des Hauptprozessors abgelegt und haben die Form des WM86 ESCAPE-Befehles. Hauptprozessor und Arithmetikprozessor sind fuer den Programmierer eine Einheit. Es stehen acht Operandenregister zu je 80 Bit zur Verfuegung. Diese koennen sowohl einzeln adressiert als auch als Stack organisiert werden. Das Zusammenspiel der beiden Prozessoren organisiert die Steuereinheit des Artithmetikprozessors. Sie stellt fest, wenn ein ESCAPE-Befehl im Hauptprozessor zur Ausfuehrung gelangt und aktiviert dann die WM87-Ausfuehrungseinheit.

Der **Befehlssatz des Arithmetikprozessors** wird in sechs Gruppen eingeteilt:

Datentransferbefehle
Die Datentransferbefehle (Lade- und Abspeicherbefehle) sind fuer alle Datenformate verwendbar. Es werden hier gleichzeitig mit dem Laden in den WM87 die Zahlen ohne Rundungsfehler in das interne 80-Bit-Gleitkommaformat gewandelt. Umgekehrt wird beim Abspeichern dieses Format in die gewuenschte Darstellungsart zurueckgewandelt. Beim Transport in diese Richtung erfolgt zunaechst das Runden entsprechend dem Zielformat. Der interne Datentyp kann ebenfalls in und aus dem Speicher transportiert werden.
Zur Adressierung der Speicheroperanden stehen alle Adressierungsarten des Hauptprozessors zur Verfuegung.

Arithmetische_Befehle
Zur dieser Gruppe gehoeren die vier arithmetischen Grundoperationen, die Wurzelberechnung und weitere Operationen, wie Vorzeichenwechsel, Multiplikation mit 2^N und Absolutwertbildung.

Vergleichsbefehle
Diese Gruppe beinhaltet Befehle zum Vergleich von Variablen, zum Pruefen ihrer Charakteristik und zum Trennen des Signifikanten (signifikante Stellen der Mantisse) vom Exponenten.

Transzententale_Befehle
Zur transzendenten Befehlsgruppe gehoeren Tangens und Arcustangens. Damit koennen alle trigonometrischen Funktionen und ihre Umkehrfunktionen fuer beliebige Argumente berechnet werden.

Konstanten
Wichtige Konstanten wie 1, 0, Pi und einige Logarithmen sind im WM87 direkt abrufbar.

Prozessorsteuerbefehle
Die Steuerung des Arithmetikprozessors wird durch die letzte Befehlsgruppe ermoeglicht. Mit diesen Befehlen koennen Ausnahmezustaende aufgrund unzulaessiger Operationen bearbeitet werden. Es koennen zur Vorbereitung verschiedener Prozesse gezielt Registerinhalte und der Prozessorstatus in den Speicher gerettet bzw. aus diesem zurueckgeholt werden.

Einem **internen Datentyp** stehen sechs **externe Datentypen** gegenueber.

Interner Datentyp: **TEMPORARY REAL**
Intern werden Gleitkommazahlen zur Basis 2 mit Vorzeichen-Betragsdarstellung verarbeitet. Jede dieser Zahlen wird mit 80 Bit kodiert. Das erste Bit ist das Vorzeichen, die naechsten 15 Bit der Exponent und die restlichen 64 Bit der Signifikant.

Externe Datentypen:
Zu den externen Datentypen gehoeren ganze Zahlen mit 16, 32 oder 64 Bit und ihre Zweierkomplementdarstellung:
WORD INTEGER, SHORT INTEGER, LONG INTEGER
sowie Gleitkommazahlen in einfacher und doppelter Genauigkeit zur Basis 2 mit Vorzeichen-Betragsdarstellung wie beim internen Format: **SHORT REAL, LONG REAL**
Bei einfach genauen Zahlen werden 8 Bit fuer den Exponenten und 23 Bit fuer den Signifikanten, bei doppelt genauen Zahlen werden 11 Bit fuer den Exponenten und 52 Bit fuer den Signifikanten verwendet.
PACKED DECIMAL
Mit dem sechsten externen Datentyp koennen ganze Dezimalzahlen mit Vorzeichen und 18 Ziffern dargestellt werden. Dabei sind je 2 Zahlen paarweise in einem Byte gepackt.

Die Arbeitsweise des WM87 ist beim Runden, bezueglich der Genauigkeit, bezueglich des Sonderoperanden Unendlich (oo) und bei moeglichen Ausnahmesituationen vom Programmierer waehlbar.

Beim Runden (gerichtetes Runden) ist einstellbar:
- zur naechsten darstellbaren Zahl (korrektes Runden),
- zur naechst kleineren Zahl (Abrunden),
- zur naechst groesseren Zahl (Aufrunden) oder
- in Richtung Ø (Abschneiden).

Bezueglich der Genauigkeit kann der Programmierer 3 Stufen waehlen: Runden nach dem 24.,
- Runden nach dem 53. oder
- Runden nach dem 64. Bit des Signifikanten.

Ergibt sich waehrend einer Berechnung der Sonderoperand ±oo, kann der Programmierer waehlen, ob bei der Weiterverarbeitung dieses Operanden das Vorzeichen beruecksichtigt werden soll oder nicht.

Der WM87 erkennt folgende Ausnahmesituationen:

Unzulaessige Operationen,
Denormalisierter Operand,
Exponentenueberlauf,
Exponentenunterlauf,
Gerundetes Ergebnis.

Tritt eine solche Ausnahme auf, kann der Programmierer eine Programmunterbrechung zulassen. Soll das Programm nicht unterbrochen werden, liefert dann der WM87 Ersatzergebnisse.

Mit dem WM87 steht dem Anwender ein Arithmetikprozessor fuer umfangreiche numerische Aufgaben zur Verfuegung. Programmiersprachen wie Assembler, PASCAL und FORTRAN unterstuetzen direkt den Baustein, einschliesslich der durch den WM87 hinzukommenden Datentypen.
Dem Programmierer stehen damit eine komplette Festkommaarithmetik und eine Gleitkommaarithmetik mit je drei Genauigkeitsstufen sowie eine 18-stellige Dezimalarithmetik zur Verfuegung.

Die hohe Rechengenauigkeit wird durch das interne Rechnen mit 64 Binaerstellen erreicht, so dass gleichzeitig durch die Laenge der internen Operandenregister eine grosse Sicherheit vor Exponentenueber- bzw. -unterlauf erreicht wird.

Typische Einsatzgebiete fuer den WM87 sind:
- Kommerzielle Datenverarbeitung,
- Graphische Terminals (3-D-Darstellung, Interpolation, Skalierung),
- Wissenschaftliche Berechnungen (erhoehte Praezision, Trigonometrie, Exponentialrechnung),

2. Mikroprozessorsystem WM86/WM87

Der Befehlsstrom im WM86/WM87-System besteht aus einer Mischung von WM86- und WM87-Befehlen. Die Execution Unit (EU) der CPU liest den Operationskode eines Befehlsbytes oder den 1. Operationskode eines Befehlswortes zur Dekodierung von der Instruction Queue (Befehlsschlange). Die Dekodiervorgaenge in der CPU werden vom WM87 an den Queue-Statusleitungen verfolgt (Queue Tracking) und koennen dadurch parallel ausgefuehrt werden.

2.1. Befehlsausfuehrung

Die ersten 5 Bits aller WM87-Befehle sind identisch. Sie kennzeichnen die Arithmetikprozessor-Befehle (ESC-Befehle).

Beispiel:

```
Befehl FSQRT:  1101 1001 1111 1010 B ≙ D9FAH
               -------
                  |
               Diese 5-Bit-Kombination (ESCAPE) kennzeichnet
               alle WM87-Befehle.
```

Alle nicht durch ESCAPE gekennzeichneten Befehle werden von der Control Unit (CU) des WM87 ignoriert. Es sind dann WM86-Befehle, die von der CPU ausgefuehrt werden.
Dekodiert die CU einen Befehl mit ESCAPE-Kode, fuehrt sie abhaengig vom Typ des Befehles diesen aus, oder sie leitet ihn weiter zur Numeric Execution Unit (NEU).
Die CPU unterscheidet bei den WM87-Befehlen zwischen Befehlen mit Speicherzugriff und ohne Speicherzugriff. Bei notwendigem Speicherzugriff berechnet die CPU die physikalische Adresse des Operanden. Ist kein Speicherzugriff erforderlich, beginnt die CPU mit dem Ausfuehren des naechsten WM86-Befehles.

Ein Speicherzugriff eines WM87-Befehles ist erforderlich, um einen Operanden in ein WM87-Stackregister zu laden bzw. umgekehrt oder um den Inhalt eines Stackregisters in den Speicher zu schreiben.

Ist der WM87-Befehl ein Speicherschreibbefehl, berechnet die CPU die physische Operandenzieladresse.

2.2. Synchronisation der Befehle

Das Ausfuehren der CPU-Befehle muss in zwei Faellen mit dem WM87-Befehlen synchronisiert werden:

- Ein WM87-Befehl, der von der NEU auszufuehren ist, muss auf seine Bearbeitung so lange warten, bis die NEU den vorangegangenen Befehl vollstaendig abgearbeitet hat.

- Bearbeitet der WM87 einen Speicherbefehl, darf die CPU so lange keine WM86-Befehle ausfuehren, bis der WM87 den Speicherzugriff abgeschlossen hat.

Der WM86-WAIT-Befehl ermoeglicht es, die CPU und den WM87 per Software zu synchronisieren. Die CPU wird so lange in Wartezustand gehalten, bis der WM87 einen Befehl in der NEU oder einen Befehl mit Speicherzugriff beendet hat. Zur Software-Synchronisation der CPU-Befehle mit den WM87-Befehlen muss jedem WM87-Befehl ein WM86-WAIT-Befehl vorangehen. Eine Reduzierung des Programmieraufwandes liefert der Assembler MASM durch automatisches Einblenden des WAIT-Befehles vor jedem WM87-Befehl.

Beispiel:

```
...
FMUL ...        ;Multiplikation
FDIV ...        ;Division
...
```

Der Assembler 86 liefert hierfuer vier Befehle:

```
...
WAIT
FMUL ...        ;Multiplikation
WAIT
FDIV ...        ;Division
```

Durch diese Befehlssequenz wird abgesichert, dass die Multiplikation beendet ist, bevor die CPU und die WM87-CU beginnen, den Divisionsbefehl zu dekodieren.

Fuehrt der WM87 einen Befehl mit Speicherzugriff aus, ist das Ausfuehren des nachfolgenden CPU-Befehles mit dem WM87-FWAIT-Befehl so lange zu stoppen, bis der WM87 seinen Speicheroperanden gelesen oder geschrieben hat. Der FWAIT-Befehl ist vom Programmierer zwischen einen WM87-Speicherzugriffsbefehl und einem nachfolgenden CPU-Befehl zu schreiben.
Der Assembler MASM verwendet fuer den WM87-FWAIT-Befehl den gleichen Kode wie fuer den WM86-WAIT-Befehl (9BH).

Beispiel:

```
FIST RECH       ;INTEGER speichern
FWAIT           ;CPU Busruhezustand bis Speichervorgang
                ;beendet
MOV AX,RECH     ;INTEGER in AX speichern
```

3. Architektur des WM87

Der Arithmetikprozessor WM87 hat 8 programmierbare Gleitkomma-Stackelemente und 7 Wortregister. In den 16 Bit langen Wortregistern werden Zustaende und Meldungen aus der Hard- und Softwareumgebung des WM87 abgelegt.

3.1. Gleitkommastack

```
 79                 64 63                          Ø Grundzustand
                                                     nach Initialisierung
 ----------------------------------------------------
|_|________________|_____________________________R3_| ST(3)
|_|________________|_____________________________R2_| ST(2)  Stackpointer
|_|________________|_____________________________R1_| ST(1)
|_|________________|_____________________________RØ_| ST(Ø)  <--TOP
|_|________________|_____________________________R7_| ST(7)
|_|________________|_____________________________R6_| ST(6)
|_|________________|_____________________________R5_| ST(5)
|_|________________|_____________________________R4_| ST(4)
 S  EXPONENT          SIGNIFIKANT

 ^
 |
Vorzeichen (Sign)
```

Die 8 Gleitkomma-Stackregister (RØ...R7) des WM87 besitzen eine Breite von 8Ø Bits und sind in 3 Felder unterteilt:

- Vorzeichen-Feld (1 Bit)
- Exponent-Feld (15 Bits)
- Signifikant-Feld (64 Bits)

Die Stackregister werden prinzipiell nur ueber den Stackpointer adressiert. Er zeigt immer zur augenblicklichen Spitze (TOP) des Stack.

Eine Schreiboperation (push) dekrementiert zuerst den Stackpointer um 1 und laedt dann einen Operanden in das neue Stacktop. Eine Leseoperation (pop) liest einen Operanden vom augenblicklichen Stacktop und inkrementiert dann den Stackpointer um 1.

3.2. Stackregister adressieren

Die Bits 13,12,11 im Statuswort (siehe Punkt 4.) geben die Nummer des Gleitkomma-Stackregisters an, auf das der Stackpointer zeigt. Das in diesen Bits angegebene Stackregister wird damit als Stacktop festgelegt. Alle anderen Stackregister werden dann relativ zum Stacktop adressiert.

```
Mnemonik:  ST(i)   mit  Ø <= i <= 7
           ---
            |
            Es wird das i-te Stackregister nach dem
            Stacktop adressiert.
```

Nach dem Initialisieren des WM87 (RESET) werden die Bits 13,12,11 im Statuswort mit ØØØB geladen. Damit liegt das Stackregister RØ als Stacktop ST(Ø) fest.

Beispiel:

```
Mit  den Bits des Statuswortes 13|12|11 wird das  Stackregister
fuenf als Stacktop festgelegt.  1  Ø  1
Mit  ST(4) z.B.  wird das 4-te Stackregister nach dem Stacktop,
das Stackregister R1 adressiert.

 79                 64 63                           Ø
  -------------------------------------------------
 |_|_______________|____________________________R3_| ST(6)
 |_|_______________|____________________________R2_| ST(5)
 |_|_______________|____________________________R1_| ST(4)
 |_|_______________|____________________________RØ_| ST(3) Stackpointer
 |_|_______________|____________________________R7_| ST(2)
 |_|_______________|____________________________R6_| ST(1)
 |_|_______________|____________________________R5_| ST(Ø) <--TOP
 |_|_______________|____________________________R4_| ST(7)
  S  EXPONENT        SIGNIFIKANT
```

4. Steuerungsumgebung

```
 15                             8    7                            Ø
 --------------------------------------------------------------------
|_B_|C3_|___ST____|C2_|C1_|CØ_|IR_|_*_|PE_|UE_|OE_|ZE_|DE_|IE_| S
|_____*_____|IC_|__RC__|__PC____|EM_|_*_|PM_|UM_|OM_|ZM_|DM_|IM_| K
|TAG(7)|TAG(6)|___________...__________________|TAG(1)|TAG(Ø)_| I
|______________________________16_LSB__________________________| B
|__4_MSB____|_Ø_|______________INSTRUCTION_OPCODE______________| B
|______________________________________________________________| D
|__4_MSB____|_________________________Ø________________________| D
```

LSB = Least Signifikant Bit
MSB = Most Signifikant Bit

S = Statuswort
K = Kontrollwort
I = Identifizierungswort
B = Befehlsadresse \
D = Datenadresse / Exception Pointers
(* = Reservierte Bits)

Der WM87 kann durch Hardware (RESET) oder durch Software (Befehl FINIT) initialisiert werden. Dabei werden das Kontrollwort, das Statuswort und das Identifizierungswort (Tag Word) mit bestimmten Bitmustern belegt. Die WM87-Register und die Exception Pointers (Ausnahmezeiger) werden nicht beeinflusst, sie enthalten unbestimmte Werte.

4.1. Statuswort

Das Statuswort gibt Auskunft ueber den augenblicklichen Betriebszustand des WM87.
Es kann mit einem WM87-Befehl (FSTSW) im Speicher abgelegt und mit CPU-Befehlen (Bit-Maskierungen) untersucht werden.

```
 15                      8  7                   0
 -------------------------------------------------
|B_|C3|ST|ST|ST|C2|C1|C0|IR|__|PE|UE|OE|ZE|DE|IE|
     |STACKTOP|              |  EXEPTION Flags  |
```

IE:	INVALID OPERATION	(ungueltige Operation)
DE:	DENORMALIZED OPERAND	(maskierter Operand)
ZE:	ZERODIVIDE	(Division durch Null)
OE:	OVERFLOW	(Ueberlauf)
UE:	UNDERFLOW	(Unterlauf)
PE:	PRECISION	(Praezision)
	(RESERVIERT)	
IR:	INTERRUPT REQUEST	(Interruptanforderung)
C0,C1,C2,C3	CONDITION CODE	(Bedingungskode)
ST:	STACKTOP-POINTER	(Zeiger auf Stackspitze)
B:	BUSY	("Besetzt"-Anzeige)

Das Statuswort ist unterteilt in:

- BUSY-Feld (Bit 15),
  Es zeigt an, ob der WM87 einen Befehl ausfuehrt (B=1) oder ob er sich im Ruhezustand befindet (B=0).

- CONDITION-Feld (Bits 14 und 10...8 = C3, C2, C1, C0)
  Verschiedene Befehle (z.B. Vergleichsbefehle) hinterlegen nach ihrer Ausfuehrung in diesem Feld einen sogenannten CONDITION-Kode. Der CONDITION-Kode wird zur Ausfuehrung bedingter Spruenge verwendet. Nach dem Ausfuehren eines WM87-Befehles, der einen CONDITION-Kode setzt, ist deshalb das Statuswort in den Speicher zu laden. Durch CPU-Befehle kann dann der CONDITION-Kode ueberprueft und abhaengig vom Pruefergebnis die notwendigen Schritte ausgefuehrt werden.

- Aktuelles STACKTOP-Feld (Bits 13...11)
  Der in diesem Feld abgelegte Kode adressiert eines der 8 Gleitkomma-Stackregister und zeigt an, welches der Stackregister R0

bis R7 die aktuelle Stackspitze (Stacktop = ST) ist.

Beachte:
Bei ST = 000B wird nach einer PUSH-Operation (*) ST dekrementiert, so dass ST zu 111B wird.
Bei ST = 111B wird nach einer POP-Operation (*) ST inkrementiert, so dass ST zu 000B wird.
(* in bestimmten Arithmetikbefehlen)

- INTERRUPT REQUEST-Feld (Bit 7)
  Stellt der WM87 eine Interruptanforderung an die CPU (bei unmaskierten EXCEPTIONS), wird IR = 1 gesetzt.

- EXCEPTION-Feld (Bits 5...0)
  Stellt die NEU fest, dass eine ungueltige Operation ausgefuehrt wurde, wird dies als Ausnahme in den EXCEPTION-Flags (PE/ UE/ OE/ ZE/ DE /IE = 1) angezeigt.

4.2. Ausnahmen (EXCEPTIONS)

Die meisten WM87-Befehle werden waehrend ihrer Ausfuehrung vom WM87 auf Ausnahmebedingungen hin untersucht.
Es werden sechs Arten von Ausnahmen unterschieden:

Ungueltige Operation (INVALID OPERATION)
Dieses Flag wird gesetzt (IE=1),
- wenn versucht wird, ein Quellregister zu lesen, das als EMPTY (leer) gekennzeichnet ist,
- wenn versucht wird, ein Zielregister zu laden, das als NOT EMPTY gekennzeichnet ist,
- wenn ein Operand ein NAN (Not-A-Number) ist,
- wenn Operanden verwendet werden, die unbestimmte Ergebnisse liefern (Division durch Null, Wurzel einer negativen Zahl).

Ueberlauf (OVERFLOW)
Dieses Flag wird gesetzt (OE=1), wenn nach einer Operation der Exponent des Ergebnisses zu gross ist und damit nicht im gewaehlten REAL-Format dargestellt werden kann.

Unterlauf (UNDERFLOW)
)as Flag wird gesetzt (UE=1), wenn nach einer Operation der Exponent des Ergebnisses zu klein ist und damit nicht im gewaehlten EAL-Format dargestellt werden kann.

aezision (PRECISION)
eses Flag wird gesetzt (PE=1), wenn nach einer Operation das ebnis nicht exakt dem Zielformat entspricht. Das Ergebnis wird n entsprechend der festgelegten Rundungsart vom WM87 gerundet.

sion durch Null (ZERODIVIDE)
versucht, einen endlichen Nichtnull-Operanden durch Null zu en, wird dieses Flag gesetzt (ZE=1).

malisiert maskiert (DENORMALIZED)
/ersuch, einen Denormal-Operanden zu verarbeiten, wird das

Flag (DE=1) gesetzt. Ein DENORMAL wird erzeugt, wenn im Ergebnis einer Operation ein Unterlauf auftritt. Der WM87 denormalisiert das Ergebnis, d.h. er liefert eine maskierte Antwort auf einen Unterlauf.

Beispiel:

Der darstellbare Exponent einer SHORT REAL-Zahl liegt zwischen -126D und +127D. Ist der Exponent nach einer Operation -130D, kann er im SHORT REAL-Format nicht dargestellt werden. Der WM87 denormalisiert den Exponenten auf das kleinste darstellbare Format -126D. Es wuerde sich kein Unterlauf ergeben, wenn das Zielformat vom Typ LONG REAL oder TEMPORARY REAL ist. In diesen Formaten sind Exponenten bis -1023D bzw. bis 16383D darstellbar.

Denormalisieren einer SHORT REAl-Zahl:

Der zu kleine Exponent ist in den darstellbaren Bereich des Zielformates zu bringen. Der Exponent wird dazu so lange inkrementiert, bis er im Zielformat dargestellt werden kann. Bei jedem Inkrementieren wird gleichzeitig der Signifikant um eine Stelle nach rechts verschoben und die fuehrende Stelle mit einer Null besetzt.

Bevor der denormalisierte Operand gespeichert wird, wird der Signifikant auf 24 Bit gerundet. Das INTEGER-Bit (Bit vor dem Gleitkomma) wird eliminiert, so dass der Signifikant mit einer Genauigkeit von 23 Bits gespeichert wird. Ein DENORMAL hat immer einen Nichtnull-Signifikanten. Zur Dezimalzahl des Exponenten -126D wird +126D addiert. Der so entstandene Exponent (biased Exponent) = 0000 0000B entspricht dem Exponten eines DENORMAL.

Denormalisiert unmaskiert

Wird versucht, einen DENORMAL-Operanden zu verarbeiten, wird das DE-Flag = 1 gesetzt. Tritt im Ergebnis einer Operation ein Unterlauf auf, und ist das Ergebnis ein Speicheroperand, gibt der WM87 eine INTERRUPT-Anforderung an die CPU (unmaskierte Antwort).

Beim Auftreten eines Unterlaufes und Speichern des Ergebnisses in einem Gleitkomma-Stackregister, addiert der WM87 die Konstante 24576D (3FFFH) zum Exponenten des Ergebnisses (siehe 5.3.). Dadurch wird der Exponent in den darstellbaren Bereich des TEMPORARY REAL-Formates gebracht. Nach der Korrektur gibt der WM87 eine Interrupt-Anforderung an die CPU (unmaskierte Anwort).

In der Interruptroutine (Exception Handler) kann die Korrekturkonstante subtrahiert werden, um dadurch den wahren Exponenten festzustellen.

4.3. Behandlung der Ausnahmen

Jngueltige Operationen, Division durch Null und denormalisierte Operanden werden vor dem Ausfuehren einer Operation erkannt. Die EXCEPTION-Flags werden in diesen Faellen gesetzt. Die Gleitkomma-Stackregister bzw. die Speicheroperanden bleiben unbeeinflusst, d.h. ungueltige Operationen erscheinen so, als waeren sie nicht ausgefuehrt worden.

Ueberlauf, Unterlauf und Praezision werden erst nach dem Ausfuehren einer Operation erkannt, d.h. die Exception-Flags werden gesetzt und die Gleitkomma-Stackregister bzw. Speicheroperanden mit den wahren Operationsergebnissen beschrieben.

Ausnahmen:
Unmaskierte Ueber- und Unterlaufausnahmen werden bei STORE- bzw. STORE- und POP-Operationen wie Ausnahmen behandelt, die vor einer Operation erkannt wurden, d.h. die Speicheroperanden werden nicht beschrieben und der Stackpointer wird nicht inkrementiert.
Treten beim Ausfuehren eines WM87-Befehles mehrere Ausnahmen gleichzeitig auf, dann gilt folgende Rangordnung:

- Denormalisiert (wenn unmaskiert)
- Ungueltige Operation
- Division durch Null
- Denormalisiert (wenn maskiert)
- Ueberlauf / Unterlauf
- Praezision

(Ausnahmen und ihre Antworten siehe Anlage)

4.4. Kontrollwort

Mit dem Kontrollwort kann der Prozessablauf des WM87 gezielt gesteuert werden. Der Befehl FLDCW laedt einen entsprechenden 16-Bit-Kode vom Speicher in das Kontrollwortregister des WM87.

```
 15                    8   7                 0
 ---------------------------------------------
|        |IC| R,C | P,C |IEM|  |PM|UM|OM|ZM|DM|IM|
 ---------------------------------------------
                              | EXCEPTION Masken|
```

```
IM:  INVALID OPERATION         (ungueltige Operation)
DM:  DENORMALIZED OPERAND      (maskierter Operand)
ZM:  ZERODIVIDE                (Division durch Null)
OM:  OVERFLOW                  (Ueberlauf)
UM:  UNDERFLOW                 (Unterlauf)
PM:  PRECISION                 (Praezision)
     (RESERVIERT)
     Kontrollbits:
IEM: INTERRUPT-ENABLE Maske
PC:  PRECISION CONTROL         (Praezisionssteuerbits)
RC:  ROUNDING CONTROL          (Rundungssteuerbits)
IC:  INFINITY CONTROL          (Unendlichsteuerung)
     (RESERVIERT)
```

Das Kontrollwort ist in folgende Felder unterteilt:

- INFINITY CONTROL-Feld (Bit 12)
  Hier kann angegeben werden, wie ein System mit REAL-Zahlen im Bereich von -oo bis +oo geschlossen dargestellt werden soll.
  Es gibt zwei Modelle zur Darstellung eines geschlossenen REAL-Zahlenbereiches: Projektiver Abschluss IC=0
  Affiner Abschluss IC=1
  Beim projektiven Modell behandelt der WM87 die Spezialgroessen

-oo und +oo als einfache vorzeichenlose Unendlichkeiten. Beim affinen Modell werden die Vorzeichen der Spezialgroessen Unendlich beruecksichtigt (siehe Punkt 5.4.).

- ROUNDING CONTROL-Feld (Bits 11, 10)
  Wenn das Ergebnis einer Operation im Zielformat nicht exakt dargestellt werden kann, ist das Ergebnis zu runden.
  Rundungsarten:
  . Runden zur naechst darstellbaren Zahl: RC=00
    Dieser Modus ist automatisch nach dem Initialisieren des WM87 eingestellt. Der vorliegende Operand mit dem Wert b wird auf den Wert auf- oder abgerundet, der b am naechsten liegt.

  . Abrunden in Richtung -oo: RC=01
    Bei diesem Modus wird immer in Richtung -oo abgerundet.

  . Aufrunden in Richtung +oo: RC=10
    Es wird immer in Richtung +oo aufgerundet.

  . Abschneiden in Richtung 0 (Chop): RC=11
    Der Modus wird fuer die INTEGER-Arithmetik verwendet, wobei der gebrochene Anteil einer REAL-Zahl stets in Richtung zur Null des Zahlenstrahles abgeschnitten wird. Es ergeben sich stets INTEGER-Zahlen.

- PRECISION CONTROL-Feld (Bits 9, 8)
  Ergebnisse koennen wahlweise mit einer Signifikantengenauigkeit
  von 24 Bits: PC=00
  Reserviert: PC=01
  53 Bits: PC=10
  64 Bits: PC=11 berechnet werden.
  Die Genauigkeit von 64 Bits ist automatisch nach dem Initialisieren des WM87 voreingestellt.

- INTERRUPT ENABLE-Maskenfeld (Bit 7)
  Tritt beim Ausfuehren eines WM87-Befehles eine Ausnahmebedingung auf, setzt der WM87 das entsprechende EXCEPTION-Flag im Statuswort. Abhaengig vom eingestellten Bitmuster im EXCEPTION-Maskenfeld des Kontrollwortes setzt der WM87 je nach Typ der Ausnahmebedingung entweder eine "maskierte" oder eine "unmaskierte" Antwort ab.
  Eine "unmaskierte" Antwort fuehrt in jedem Fall zu einer Interruptanforderung des WM87 an die CPU. Dazu setzt der WM87 das INTERRUPT REQUEST-Bit IR im Statuswort auf 1.
  Die Interruptanforderung des WM87 kann nur gezielt gesperrt oder freigegeben werden, indem mit dem Befehl FLDCW im Kontrollwort das Flag IEM=1 (Interrupt gesperrt, Maskierung aktiv) oder IEM=0 (Interrupt freigeben) gesetzt wird.

- EXCEPTION-Maskenfeld (Bits 5, 4, 3, 2, 1, 0)
  Mit dem Befehl FLDCW kann dieses Feld mit einem bestimmten Bitmuster belegt werden. Dadurch kann der WM87 zu einer "maskierten" (EXCEPTION-Maskenflag=1) oder "unmaskierten" (EXCEPTION-Maskenflag=0) Antwort veranlasst werden. Jeder moeglichen Ausnahmebedingung ist ein Flagbit im EXCEPTION-Maskenfeld zugeordnet. Bei Ausnahmebedingungen prueft der WM87

den logischen Zustand des entsprechenden Flags.
Eine "unmaskierte" Anwort bedeutet eine Interruptanforderung des WM87 an die CPU. Akzeptiert die CPU diese Anforderung, wird eine Interruptprozedur aufgerufen, in der ein vom Anwender geschriebener "Exception Handler" den auftretenden Fehler identifiziert, anwenderabhaengige Aktionen ausfuehrt und den Ruecksprung zum Unterbrechungspunkt veranlasst.
Eine "maskierte" Antwort (IEM=1 und Maskenfeldbits gesetzt) bedeutet eine konkrete Reaktion des WM87 auf alle Ausnahmen (siehe Anlage).

4.5. Identifizierungswort (Tag Word)

Jedem Stackregister des WM87 ist im Identifizierungswort je ein 2-Bit TAG-Feld zugeordnet. Der Inhalt des TAG-Feldes enthaelt den TAG-Kode, der den im Stackregister gespeicherten Datentyp charakterisiert.

15				7			0
TAG(7)	TAG(6)	TAG(5)	TAG(4)	TAG(3)	TAG(2)	TAG(1)	TAG(0)

TAG-Kode:
00 = Gueltig (NORMAL oder UNNORMAL)
01 = Wahre Null
10 = SPECIAL (NAN, ∞ oder DENORMAL)
11 = Leer (EMPTY)

Befehle, die nach Ausfuehren einer Operation den Stackpointer inkrementieren, besetzen das TAG-Feld des vorherigen Stackregisters mit dem Kode fuer EMPTY. Stackregister, die als EMPTY gekennzeichnet sind, duerfen nur beschrieben, aber nicht gelesen werden.

4.6. Exception-Pointers

Wird im WM86/WM87-Prozessorsystem ein WM87-Befehl ausgefuehrt, legt der WM86 die Befehlsadresse (Instruction Pointer) auf den Adressbus und liest den Befehlskode (Instruction-Operationskode) vom Programmspeicher. Beide Komponenten werden in die entsprechenden EXCEPTION Pointer-Register der CU im WM87 geladen.

Fuehrt der WM87 eine Operation mit einem Speicheroperanden aus, berechnet die CPU die physikalische Adresse des WM87-Operanden. Die Speicheroperandenadresse wird zusaetzlich in die entsprechenden EXCEPTION Pointer-Register des WM87 geladen.

Eine "EXCEPTION-Prozedur" kann diese Zeiger in den Speicher schreiben, sie untersuchen und dadurch die notwendigen Informationen ueber den Befehl erhalten, der die Exception verursacht hat.

4.7. Interruptverarbeitung

Tritt beim Ausfuehren eines WM87-Befehles eine Ausnahmebedingung auf, kann der WM87 als "unmaskierte Antwort" von der CPU einen Interrupt anfordern.
Ist die Exception maskiert, gibt der WM87 eine entsprechende "maskierte Antwort", setzt aber nicht das INTERRUPT REQUEST-Bit (IR) im Statuswort. Bei einer unmaskierten Exception setzt der WM87 IR = 1.

Das INTERRUPT REQUEST-Bit im Statuswort bleibt so lange gesetzt, bis es explizit durch die Befehle FNCLEX, FNSAVE oder FNINIT geloescht wird.
Die Interruptprozedur (Exception Handler), die durch die Interruptanforderung des WM87 aufgerufen wird, muss das INTERRUPT REQUEST-Bit loeschen, bevor der Ruecksprung in das WM87-Programm erfolgt. Anderenfalls wird nach dem Ruecksprung ein erneuter Interrupt erzeugt (Endlosschleife).

4.8. Exception Handler

Ein vom Anwender geschriebener Exception Handler hat die Form einer WM86-Prozedur. Er ist, abhaengig von der Anwendung, unterschiedlich und enthaelt meist folgende grundsaetzliche Schritte:
Die beim Auftreten der Ausnahme existierende WM87-Umgebung (Kontrollwort, Statuswort, Identifizierungswort, Exception Pointers) wird mit dem Befehl FSTENV gespeichert. Das Exception-Bit im Statuswort ist zu loeschen. Die Interruptanforderung zur CPU ist wieder freizugeben.

Beachte:
In einem Exception Handler duerfen keine WM87-Befehle mit vorangestelltem WAIT-Kode (oder FWAIT-Befehl) verwendet werden, wenn die Interruptanforderungen zur CPU gesperrt sind.

Die Ausnahme ist durch Pruefen des Status- und Kontrollwortes in der gespeicherten WM87-Umgebung zu identifizieren.
Anwenderabhaengige Aktionen sind auszufuehren und wieder zum Unterbrechungspunkt zurueckzuspringen, um die normale Befehlesausfuehrung fortzusetzen.

Moegliche anwenderabhaengige Aktionen sind:
- Exception-Zaehler inkrementieren (zur Ausgabe an Bildschirm oder Drucker),
- Diagnoseinformationen (z.B. WM87-Umgebung und Stackregister) an Bildschirm oder Drucker senden,
- Abbruch der Berechnung, durch die die Ausnahme verursacht wurde,
- Abbruch der weiteren Befehlsausfuehrung,
- Verwenden der Exception Pointers, zum Generieren eines Befehles, von dem keine Ausnahme zu erwarten ist, und diesen ausfuehren,
- Diagnosewert im Ergebnis speichern (eine NAN) und Berechnung fortsetzen.

5. Datentypen und Datenformate

Der WM87 kennt 7 numerische Datentypen, die in 3 Klassen eingeteilt sind:

- BINARY INTEGER: WORD INTEGER, SHORT INTEGER, LONG INTEGER
- PACKED DECIMAL INTEGER
- BINARY REAL: SHORT REAL, LONG REAL, TEMPORARY REAL

```
WORD INTEGER       -------------
                  |S|   WERT    |
                  |_|___________|
                  15            0

SHORT INTEGER      ----------------------
                  |S|       WERT         |
                  |_|____________________|
                  31                     0

LONG INTEGER       ----------------------------------------
                  |S|               WERT                   |
                  |_|______________________________________|
                  63                                       0

PACKED DECIMAL     -------------------------------------------------
                  |S| X |                WERT                       |
                  |   |d17|d16|d15|          ...        |d2|d1|d0 |
                  |_|___|___|___|___|___             ___|__|__|___|
                  79  72                                 8  4  0

SHORT REAL         ------------------------------
                  |S| BIASED  |                  |
                  | |EXPONENT | SIGNIFIKANT      |
                  |_|_________|__________________|
                  31          23                 0
                              ^
                              I (Integerbit bleibt als implizite
                                 Angabe unberuecksichtigt)

LONG REAL          ----------------------------------------
                  |S|  BIASED   |                          |
                  |_|_EXPONENT__|_______SIGNIFIKANT________|
                  63           52                          0
                                ^
                                I (siehe SHORT REAL)

TEMPORARY REAL     ------------------------------------------------
                  |S|  BIASED      | _ |                          |
                  |_|__EXPONENT____|_I_|_____SIGNIFIKANT__________|
                  79              64  63                          0
```

S = Sign (Vorzeichen), I = Integerbit, X = nicht genutzt

5.1. Binary Integer

Die INTEGER-Formate sind bis auf ihre Laenge identisch.

Datentyp	Bits	Dezimaler Zahlenbereich
WORD INTEGER	16	$-32\ 768 <= x <= 32\ 767$
SHORT INTEGER	32	$-2*10^9 <= x <= +2*10^9$
LONG INTEGER	64	$-9*10^{18} <= x <= +9*10^{18}$

Das Vorzeichen einer Zahl wird mit Bit 15 bei WORD INTEGER, Bit 31 bei SHORT INTEGER und Bit 63 bei LONG INTEGER bestimmt. Das Vorzeichenbit ist 0 fuer positive und 1 fuer negative Zahlen. Negative Zahlen werden im Zweierkomplement dargestellt.

Das WM87 WORD INTEGER-Format ist identisch mit dem 16-Bit vorzeichenbehafteten INTEGER Datenformat des WM86.
Die INTEGER-Operanden sind so gespeichert, dass das Vorzeichenbit immer im hoeherwertigen Adressbyte liegt.

Der WM87 liefert die INTEGER INDEFINITE-Zahl, wenn sich als Operationsergebnis die groesste negative Zahl ergibt oder als Antwort auf eine maskierte INVALID OPERATION-Ausnahme.
Wird diese Zahl als Quelloperand z.B. bei einem Integerladebefehl verwendet, interpretiert sie der WM87 als groesste negative Zahl im Format:
INTEGER INDEFINITE: -2^{15}, -2^{31}, -2^{63}

5.2. Packed Decimal Integer

Datentyp	Bits	Dezimaler Zahlenbereich
PACKED DECIMAL	80	$-10^{18}+1 <= x <= 10^{18}-1$ 18 Stellen 18 Stellen

Dezimale INTEGER-Zahlen werden in gepackter dezimaler Form gespeichert. Jeweils 2 dezimale Stellen im Zahlenbereich 0...9H werden in ein Byte "gepackt".
Die Bits 72...78 haben keine Bedeutung, das Bit 79 legt das Vorzeichen fest (0 = positiv, 1 = negativ). Negative Zahlen unterscheiden sich von positiven nur durch das Vorzeichen.
Als Antwort auf eine maskierte INVALID OPERATION-Ausnahme ergibt sich die PACKED DECIMAL INDEFINITE-Zahl. Sie kann mit dem Befehl FBSTP im Speicher abgelegt werden. Wird diese Zahl mit dem Befehl FBLD geladen, entsteht ein undefiniertes Ergebnis.
PACKED DECIMAL INDEFINITE: $-10^{18}+1$

5.3. Real

Datentyp	Bits	Dezimaler Zahlenbereich
SHORT REAL	32	$8.43*10^{-37} <= \|x\| <= 3.37*10^{38}$
LONG REAL	64	$4.19*10^{-307} <= \|x\| <= 1.67*10^{308}$
TEMPORARY REAL	80	$3.4*10^{-4932} <= \|x\| <= 1.2*10^{4932}$

REAL-Zahlen werden in einem binaeren Format gespeichert, das aus dem SIGNIFIKANT-Feld, dem BIASED EXPONENT-Feld und dem SIGN-Feld (Vorzeichenfeld) besteht.

Formel zur Darstellung einer REAL-Zahl (Gleitkommazahl):

$$(-1)^S \quad * \quad (2^{E-Bias}) \; * \; (f_{N...}f_0)$$

Signifikant
N = 22 (SHORT REAL)
N = 51 (LONG REAL)
N = 62 (TEMPORARYREAL)

Wahrer Exponent
Bias = 127D (7FH) (SHORT REAL)
Bias = 1023D (3FFH) (LONG REAL)
Bias = 16383D (3FFFH) (TEMPORARY REAL)
E = Exponent
Bias = Verschiebekonstante

Sign
S = 0: positiv
S = 1: negativ

Beim REAL-Operanden im Speicher liegt das Vorzeichenbit immer im hoechstwertigen Adressbyte.

5.4. Spezielle Real-Zahlen

DENORMALs

Ein DENORMAL ist eine maskierte Anwort auf eine UNDERFLOW-Ausnahme (Unterlauf). DENORMALs werden durch Unterlauf ihrer Exponenten identifiziert. Der Exponent eines DENORMALs ist identisch mit dem einer Null. Ein DENORMAL hat einen Nicht-Null-Signifikanten. DENORMALE entsprechen dem Bitstring 00...00 im Exponenten.

NANs (Not-A-Number)

NANs koennen vorzeichenbehaftet sein (SIGN = 0 oder 1) und werden durch ihre Exponenten identifiziert. Der Exponent entspricht dem Bitstring 11...11B. Der Signifikant kann einen beliebigen Bitstring enthalten, ausser 10...00B. Diese REAL-Zahl wird als REAL INDEFINITE bezeichnet, ihr Vorzeichen ist negativ.

Alle anderen NANs repraesentieren Werte, die vom Programmierer kodiert werden koennen. Wird ein NAN als Operand verarbeitet, zeigt dies der WM87 als ungueltige Operation an und liefert als maskierte Antwort das NAN als Operarionsergebnis. Werden zwei NANs als Operanden verarbeitet, liefert der WM87 das NAN mit dem groesseren Absolutwert als Ergebnis.

INFINITIES (Unendlich)
Die vorzeichenbehaftete Darstellung von Unendlich (oo) wird unterstuetzt. Der verschobene Exponent enthaelt den Bitstring 11...11B, der Signifikant den Bitstring 00...00B. Das Vorzeichenfeld kann 0 (+oo) oder 1 (-oo) enthalten.
INFINITES koennen vom Programmierer kodiert oder vom WM87 als maskierte Antwort auf eine OVERFLOW- (Ueberlauf) bzw. ZERODIVIDE- (Nulldivision) Ausnahme gebildet werden.
Werden INFINITES als Operanden verwendet, ist das Verhalten des WM87 davon abhaengig, wie das INIFINITY-Kontrollfeld (IC) im Kontrollwort gesetzt ist.

Verhalten von INFINITES als Operanden

Operation	IC = 0 Projektives Ergebnis	IC = 1 Affines Ergebnis
ADDITION		
+oo plus +oo	INVALID OPERATION	+oo
-oo plus -oo	INVALID OPERATION	-oo
+oo plus -oo	INVALID OPERATION	INVALID OPERATION
-oo plus +oo	INVALID OPERATION	INVALID OPERATION
±oo plus ±X	#oo	#oo
±X plus ±oo	#oo	#oo
SUBTRAKTION		
+oo minus-oo	INVALID OPERATION	+oo
-oo minus+oo	INVALID OPERATION	-oo
+oo minus+oo	INVALID OPERATION	INVALID OPERATION
-oo minus-oo	INVALID OPERATION	INVALID OPERATION
±oo minus±X	#oo	#oo
±X minus±oo	≠oo	≠oo
MULTIPLIKATION		
±oo * ±oo	θoo	θoo
±oo * ±Y	θoo	θoo
±0 * ±oo,±oo#±0	INVALID OPERATION	INVALID OPERATION
DIVISION		
±oo / ±oo	INVALID OPERATION	INVALID OPERATION
±oo / ±X	θoo	θoo
±X / ±oo	θ0	θ0
FSQRT		
-oo	INVALID OPERATION	INVALID OPERATION
+oo	INVALID OPERATION	+oo

Operation	Projektives Ergebnis	Affines Ergebnis
FPREM		
±oo rem ±oo	INVALID OPERATION	INVALID OPERATION
±oo rem ±X	INVALID OPERATION	INVALID OPERATION
±Y rem ±oo	#Y	#Y
±0 rem ±oo	#0	#0
FRNDINT		
±oo	#oo	#oo
FSCALE		
±oo scaled by ±oo	INVALID OPERATION	INVALID OPERATION
±oo scaled by ±X	#oo	#oo
±0 scaled by ±oo	#0	#0
±Y scaled by ±oo	INVALID OPERATION	INVALID OPERATION
FXTRACT		
±oo	INVALID OPERATION	INVALID OPERATION
VERGLEICH		
±oo : ±oo	A = B	-oo < +oo
±oo : ±Y	A ? B (und) INVALID OPERATION	-oo < Y < +oo
±oo : ±0	A ? B (und) INVALID OPERATION	-oo < 0 < +oo
FTST		
±oo	A ? B (und) INVALID OPERATION	#oo

Erklaerung:
X = Null oder Nicht-Null-Operand
Y = Nicht-Null-Operand
\# = Vorzeichen des Original-Operanden
≠ = Vorzeichen entspricht dem Komplement des Vorzeichens des Original-Operanden
⊖ = Vorzeichen entspricht der Exclusiv-Oder-Verknuepfung der Vorzeichen von den Original-Operanden (+ bei gleichem Vorzeichen, - bei verschiedenen Vorzeichen).

NULLEN
Die REAL- und PACKED DECIMAL-Datentypen unterstuetzen vorzeichenbehaftete Nullen. BINARY INTEGER-Datentypen unterstuetzen eine Null mit positivem Vorzeichen. Vorzeichenbehaftete Nullen verhalten sich so, als ob sie eine vorzeichenlose Groesse sind. Mit dem Befehl FXAM kann das Vorzeichen einer Null bestimmt werden. Die vorzeichenbehafteten Nullen der REAL- und PACKED DECIMAL-Typen werden als "wahre Nullen" bezeichnet.

Behandlung von Null-Operanden, Erzeugen von "wahren Nullen durch Nicht-Null-Operanden (bezeichnet mit X bzw. Y:

Operation/ Operanden	Ergebnis	Operation/ Operanden	Ergebnis
FLD, FBLD (1)		Division	
+0	+0	±0 / ±0	INVALID OPERAT.
-0	-0	±X / ±0	ZERODIVIDE
FILD (2)		+0/+X,-0/-X	+0
+0	+0	+0/-X,-0/+X	-0
FST, FSTP		-X/-Y,+X/+Y	+0,UNDERFLOW(8)
+0	+0	-X/+Y,+X/-Y	-0,UNDERFLOW(8)
-0	-0	FPREM	
+X(3)	+0	±0rem±0	INVALID OPERAT.
-X(3)	-0	±Xrem±0	INVALID OPERAT.
FBSTB		+0rem+X,+0rem-X	+0
+0	+0	-0rem+X,-0rem-X	-0
-0	-0	+Xrem+Y,+Xrem-Y	+0 (9)
FIST, FISTP		-Xrem-Y,-Xrem+Y	-0 (9)
+0	+0	FSQRT	
-0	+0	-0	-0
+X (4)	+0	+0	+0
-X (4)	+0		
Addition		Vergleich	
+0plus +0	+0	±0:+X	A < B
-0plus -0	-0	±0:±0	A = B
+0plus -0,-0plus +0	#0 (5)	±0:-X	A > B
-Xplus +X,+Xplus -X	#0 (5)	FTST	
±0plus ±X,±Xplus ±0	≠0 (6)	±0	Null
Subtraktion		FCHS	
+0minus-0	+0	+0	-0
-0minus+0	-0	-0	+0
+0minus+0,-0minus-0	#0 (5)	FABS	
+Xminus+X,-Xminus-X	#0 (5)	±0	+0
±0minus±X,±Xminus±0	≠0 (6)	F2XM1	
Multiplikation		+0	+0
+0*+0,-0*-0	+0	-0	-0
+0*-0,-0*+0	-0	FRNDINT	
+0*+X,+X*+0	+0	+0	+0
+0*-X,-X*+0	-0	-0	-0
-0*+X,+X*-0	-0	FXTRACT	
-0*-X,-X*-0	+0	+0	Beide +0
+X*+Y,-X*-Y	+0	-0	Beide -0
	UNDERF.(7)		
+X*-Y,-X*+Y	-0,		
	UNDERF.(7)		

Erklaerungen:

(1) Arithmetische- und Vergleichsoperationen mit REAL-Speicheroperanden interpretieren die Vorzeichen der Speicheroperanden in gleicher Weise.

(2) Arithmetische- und Vergleichsoperationen mit BINARY INTEGER interpretieren das Vorzeichen der INTEGER-Zahl in gleicher Weise.

(3) DENORMALs, deren Signifikant aus Nullen besteht, werden als Nullen im SHORT- oder LONG-REAL-Format gespeichert.

(4) Kleine Werte (|X|<1), die im INTEGER-Format zu speichern sind, werden nach Null gerundet.

(5) Das Vorzeichen ist durch die Rundungsart bestimmt.
= + fuer Runden zum naechsten Wert, Aufrunden oder in Richtung Null runden
= - fuer Abrunden

(6) ≠ = Vorzeichen von X

(7) Sehr kleine Werte vor X und Y koennen nach dem Runden des wahren Ergebnisses eine Null erzeugen.
Der WM87 meldet zur Warnung Underflow, um anzuzeigen, dass sich eine Null durch Nicht-Null-Operanden ergeben hat.

(8) Sehr kleines X und sehr grosses Y koennen nach dem Runden des wahren Ergebnisses eine Null erzeugen.
Underflow wird gemeldet, um anzuzeigen, dass sich eine Null durch Nicht-Null-Operanden ergeben hat.

(9) Wenn Y ohne Rest in X enthalten ist.

UNNORMALs

Ein UNNORMAL ist die Antwort auf eine maskierte UNDERFLOW-Ausnahme. Es existiert nur im TEMPORARY REAL-Format.
Der Exponent kann den gleichen Bitstring wie ein NORMAL (Operand, der im darstellbaren Bereich des Zielformates liegt) enthalten.
Das UNNORMAL unterscheidet sich vom TEMPORARY NORMAL durch das INTEGER-Bit im Signifikanten. Es ist beim UNNORMAL immer 0.
Ein UNNORMAL kann als Originaloperand vom WM87 verarbeitet werden oder wird aus einem DENORMAL als maskierte Antwort aufgrund einer Ausnahme gebildet.

Behandlung von UNNORMAL-Operanden durch den WM87-Befehlssatz:

Operation	Ergebnis
Addition/Subtraktion	Beim Verknuepfen einer kleinen UNNORMAL-Zahl mit einer grossen NORMAL-Zahl, ergibt sich ein normalisiertes Ergebnis. Umgekehrt ist das Ergebnis UNNORMAL.
Multiplikation	Ist ein Operand ein UNNORMAL, ist das Ergebnis auch ein UNNORMAL.
Division (mit UNNORMAL Divident)	Das Ergebnis ist ein NORMAL

Operation	Ergebnis
FPREM (mit UNNORMAL Divident)	Das Ergebnis ist normalisiert.
DIVISION / FPREM (UNNORMAL Divisor)	Eine ungueltige Operation wird angezeigt
Compare / FTST	Vor dem Vergleich wird so weit wie moeglich normalisiert.
FRNDINT	Vor dem Runden wird so weit wie moeglich normalisiert.
FSQRT	Es wird eine ungueltige Operation angezeigt.
FST, FSTP (SHORT / LONG Ziel)	Liegt der Wert oberhalb der UNDERFLOW-Grenze, wird eine ungueltige Operation, im anderen Fall UNDERFLOW angezeigt.
FSTP (TEMPORARY REAL Ziel)	Speicherung wie ueblich
FIST, FISP, FBSTP	Eine ungueltige Operation wird angezeigt
FLD	Laden wie ueblich
FXCH	Vertauschen wie ueblich
Transzendente Befehle	Undefiniert, da Operanden normalisiert sein muessen und nicht geprueft werden.

PSEUDO-NULLEN

Sie existieren nur im TEMPORARY REAL-Format. Eine PSEUDO-NULL ist ein UNNORMAL, dessen Signifikant den Bitstring Ø‸ØØ...ØØB enthaelt. Sein verschobener Exponent aber nicht Null ist.
Ein PSEUDO-NULL-Resultat ergibt sich, wenn zwei UNNORMALs multipliziert werden, deren Signifikanten zusammen mehr als 64 fuehrende Nullen enthalten.
PSEUDO-NULL-Operanden verhalten sich wie UNNORMALs.
Ausnahmen, bei denen Ergebnisse wie bei "wahren Nullen" geliefert werden, sind Vergleichs- und Testbefehle, FRNDINT und Division, bei der der Divident entweder eine "wahre Null" oder eine PSEUDO-NULL ist (Divisor ist eine PSEUDO-NULL).

5.5. Zahlenbereich von REALs

Da der WM87 innerhalb seines endlichen Zahlenbereiches nicht alle Zahlen darstellen kann, gibt es zwischen zwei darstellbaren benachbarten Zahlen eine Luecke. Faellt ein Ergebnis einer Operation in diese Luecke, wird das Ergebnis gerundet.
Zwischen 2 und 4 gibt es so viele darstellbare Zahlen wie zwischen 65 536 und 131 072, weil die dazwischenliegenden Zahlen nacheinander als Potenzen zur Basis 2 gebildet werden. Je groesser eine darstellbare Zahl ist, um so groesser ist auch die Luecke zur darstellbaren Nachbarzahl.

Alle INTEGER-Zahlen im Bereich von $\pm 2^{64}$ sind exakt darstellbar.

Zahlenbereich aller SHORT und LONG REALS

```
 -----------------------------------------------------------------
|                            |      | Biased    |Signifikant*|
|   Typ                      | Sign | Exponent  |  ^ff...ff  |
|============================|======|===========|============|
|   |                        |  0   |  11...11  |  11...11   |
|   |                        |  .   |     .     |     .      |
|   |         NANs           |  .   |     .     |     .      |
|   |                        |  .   |     .     |     .      |
| P |                        |  0   |  11...11  |  00...01   |
| O |------------------------|------|-----------|------------|
| S |         oo             |  0   |  11...11  |  00...00   |
| I |---|--------------------|------|-----------|------------|
| T |   | NORMALs            |  0   |  11...10  |  11...11   |
| I |   |                    |  .   |     .     |     .      |
| V |   |                    |  .   |     .     |     .      |
| E | R |                    |  .   |     .     |     .      |
|   | E |                    |  0   |  00...01  |  00...00   |
|   | A |--------------------|------|-----------|------------|
|   | L | DENORMALs          |  0   |  00...00  |  11...11   |
|   | s |                    |  .   |     .     |     .      |
|   |   |                    |  .   |     .     |     .      |
|   |   |                    |  0   |  00...00  |  00...01   |
|   |   |--------------------|------|-----------|------------|
|   |   | NULL               |  0   |  00...00  |  00...00   |
|---|---|--------------------|------|-----------|------------|
|   |   | NULL               |  1   |  00...00  |  00...00   |
|   |   |--------------------|------|-----------|------------|
|   |   | DENORMALs          |  1   |  00...00  |  00...01   |
|   |   |                    |  .   |     .     |     .      |
|   | R |                    |  .   |     .     |     .      |
| N | E |                    |  .   |     .     |     .      |
| E | A |                    |  1   |  00...00  |  11...11   |
| G | L |--------------------|------|-----------|------------|
| A | s | NORMALs            |  1   |  00...01  |  00...00   |
| T |   |                    |  .   |     .     |     .      |
| I |   |                    |  .   |     .     |     .      |
| V |   |                    |  .   |     .     |     .      |
| E |   |                    |  1   |  11...10  |  11...11   |
|   |------------------------|------|-----------|------------|
|   |         oo             |  1   |  11...11  |  00...00   |
|   |------------------------|------|-----------|------------|
|   |                        |  1   |  11...11  |  00...01   |
|   |                        |  .   |     .     |     .      |
|   |           -------------|------|-----------|------------|
|   |NANs |INDEFINITE        |  1   |  11...11  |  10...00   |
|   |           -------------|------|-----------|------------|
|   |                        |  .   |     .     |     .      |
|___|________________________|__1___|__11...11__|__11...11___|
                       SHORT: |<- 8Bits->|<-23 Bits-->|
                       LONG : |<-11Bits->|<-52 Bits-->|
```

* Das INTEGER-Bit ist implizit vorhanden, also nicht gespeichert.

Zahlenbereich aller TEMPORARY REALs

Typ			Sign	Biased Exponent	Signifikant* I‸ff...ff
POSITIVE	NANs		0	11...11	111...11
			.	.	.
			.	.	.
			.	.	.
			0	11...11	100...01
	oo		0	11...11	100...00
	REALs		0	11...10	NORMALs 111...11
			.	.	.
			.	.	.
			.	.	.
					100...00
					UNNORMALs 011...11
			.	.	.
			.		.
			0	00...01	000...00
					DENORMALs
			0	00...00	011...11
			.	.	.
			.	.	.
			0	00...00	000...01
		NULL	0	00...00	000...00
	REALs	NULL	1	00...00	000...00
NEGATIVE					DENORMALs
			1	00...00	000...01
			.	.	.
			.	.	.
			1	00...00	011...11
			1	00...01	UNNORMALs 000...00
			.	.	.
			.	.	.
			.	.	.
					011...11
					NORMALs 100...00
					.
					.
			1	11...10	111...11
	oo		1	11...11	100...00
	NANs		1	11...11	100...00
			.	.	.
		INDEFINITE	1	11...11	110...00
			.	.	.
			1	11...11	111...11
				<-15Bits->	<-64 Bits-->

*Das INTEGER-Bit des Signifikanten wird im TEMPORARY REAL-Format gespeichert.

6. Datendefinition, Adressierung

6.1. Definieren von Daten

Um fuer die WM87-Variable und -Konstante Speicherplaetze zu reservieren, bietet der WM86-Assembler MASM einige Direktiven. Den mit diesen Direktiven vereinbarten Variablen ordnet er definierte Datentypen zu.

Direktiven zur Speicherplatzzuteilung:

Direktive	Interpretation	WM87 Datentypen	Speicherplatz in Bytes
DW	DEFINE WORD	WORD INTEGER	2
DD	DEFINE DOUBLEWORD	SHORT INTEGER, SHORT REAL	4
DQ	DEFINE QUADWORD	LONG INTEGER, LONG REAL	8
DT	DEFINE TENBYTE	PACKED DECIMAL, TEMPORARY REAL	10

Der in einem WM87-Befehl kodierte Typ einer Variablen wird vom Assembler ueberprueft, um abzusichern, dass er zum Befehl kompatibel ist.
Ist es notwendig, in einem Befehl einen Operanden zu verwenden, dessen Typ nicht deklariert ist (z.B. FIADD [BX], muss der Assembler ueber den Typ des adressierten Operanden durch einen "Typ-Operator" informiert werden (z.B. FIADD DWORD PTR [BX].

Typ-Operatoren:

Typ-Operator	Interpretation
WORD PTR	Zeiger auf einen WORD INTEGER-Operanden
DWORD PTR	Zeiger auf einen SHORT INTEGER- oder SHORT REAL-Operanden
QWORD PTR	Zeiger auf einen LONG INTEGER- oder LONG REAL-Operanden
TBYTE PTR	Zeiger auf einen PACKED DECIMAL- oder TEMPORARY REAL-Operanden

Beachte:
Der Assembler prueft nicht den Operandentyp, der in Prozessorsteuerbefehlen verwendet wird.

6.2. Adressieren der WM87-Operanden

Aufbau eines WM87-Befehles mit Speicherzugriff:

```
7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
----------------------------------------------------------------
| ESCAPE   |X X X|MOD|X X X| R/M |Displacem.-LOW |Displacem.-HIGH|
|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|
```

Das 16-Bit-DISPLACEMENT ist eine weitere Komponente zur Bildung der effektiven Adresse eines WM87-Speicheroperanden.
Durch entsprechende Bitkombinationen im Modusfeld (MOD) und Register/Memory-Feld (R/M) kann auf WM87-Speicheroperanden durch sechs verschiedene Adressierungsarten zugegriffen werden.
Die Laenge des Befehles variiert zwischen zwei und vier Byte, wobei das dritte und vierte Byte als 8- oder 16-Bit-Displacement verwendet wird.
Die CPU verwendet bis zu drei Komponenten, um die effektive Adresse eines WM87-Operanden zu berechnen:

Displacement im WM87-Befehl
Basisregister (Inhalt von BX oder BP)
Indexregister (Inhalt von SI und DI

Es werden sechs Speicheradressierungsarten generiert, wenn diese drei Komponenten in unterschiedlicher Weise kombiniert werden.
Diese Adressierungsarten sind notwendig, um auf die verschiedenen Typen von Speicherdaten zugreifen zu koennen.

Adressierungsart	Offset-Berechnung
Direkt	EA = DISP
Register indirekt	EA = [BX] oder [BP] oder [SI] oder [DI]
Basisadressmodus	EA = ([BX] oder [BP]) + DISP
Indexadressmodus	EA = ([SI] oder [DI]) + DISP
Basis-u.Indexadr. modus kombiniert	EA = ([BX] oder [BP]) + ([SI] oder [DI])
Basis-u.Indexadr. modus kombiniert mit Displacement	EA = [(BX) oder (BP)] + [(SI) oder (DI)] + DISP

Auswahl der Adressierungsarten durch die r/m- und mod-Felder im Befehlescode:

r/m	mod = 00	mod = 01 **	mod = 0
000	[BX]+[SI]	[BX]+[SI]+ DISP	[BX]+[SI]+ DISP
001	[BX]+[DI]	[BX]+[DI]+ DISP	[BX]+[DI]+ DISP
010	[BP]+[SI]	[BP]+[SI]+ DISP	[BP]+[SI]+ DISP
011	[BP]+[DI]	[BP]+[DI]+ DISP	[BP]+[DI]+ DISP

r/m	mod = 00	mod = 01 **	mod = 0
100	[SI]	[SI]+ DISP	[SI]+ DISP
101	[DI]	[DI]+ DISP	[DI]+ DISP
110	DISP *	[BP]+ DISP	[BP]+ DISP
111	[BX]	[BX]+ DISP	[BX]+ DISP
		nur DISP.LOW	DISP.LOW und DISP.HIGH

Erklaerung:

* Die Bitkombinationen r/m = 110 und mod = 00 bilden eine Ausnahme.
In diesem Fall bilden die beiden auf das Adressmodusbyte folgenden DISPLACEMENT-Bytes die effektive 16-Bit-Adresse.

** Steht im mod-Feld die Bitkombination 01 (mod = 01), ist DISPLACEMENT-HIGH nicht vorhanden. In diesem Fall wird DISPLACEMENT-LOW abhaengig vom Wert des MSB (Bit 7) der CPU auf 16 Bit vorzeichenbehaftet erweitert.

WM87-Befehl ohne Speicherzugriff

Diese Befehle werden verwendet, um Operationen zwischen den Stackregistern des WM87 durchzufuehren.
Dabei wird das Stackregister, in dem der Quell- bzw. der Zieloperand steht, durch eine entsprechende Bitkombination im ST(i)-Feld des Befehles adressiert.
Der Befehl ist immer zwei Byte lang.

Aufbau eines WM87-Befehles ohne Speicherzugriff:

```
 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
 -------------------------------
| ESCAPE   |X X X|MOD|X X X|ST(i)|
|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|
```

Das ST(i)-Feld entspricht dem R/M-Feld. Durch entsprechende Bitkombinationen wird das i-te Stackregister nach dem Stacktop adressiert.
Im MOD-Feld steht die Bitkombination 11. Dadurch wird das R/M-Feld als ST(i)-Feld behandelt.

7. Befehle des Arithmetikprozessors

7.1. Datentransferbefehle

Die Datentransferbefehle transportieren Operanden zwischen den Stackelementen bzw. zwischen dem Stacktop und dem Datenspeicher.

Uebersicht:

	REAL transfers
FLD	load REAL
FST	store REAL
FSTP	store REAL and pop
FXCH	exchange registers
	INTEGER transfers
FILD	INTEGER load
FIST	INTEGER store
FISTP	INTEGER store and pop
	PACKED DECIMAL transfers
FBLD	PACKED DECIMAL (BCD) load
FBSTP	PACKED DECIMAL (BCD) store and pop

Nach Ausfuehren eines Transferbefehles wird das WM87 Tag Word aktualisiert.

7.1.1. REAL in Stacktop laden

Befehl:	FLD	load REAL
Schreibweise:	FLD quelle	

Exception-Flags: IE, DE

Operandenkombinationen:

Typ	quelle
1	ST(i)
2	mem-op

Befehlswirkung:

Ein REAL-Operand wird an die Spitze des Gleitkomma-Stack geladen. Dabei wird der Stackpointer zunaechst dekrementiert und danach der Quelloperand an den neuen Stacktop geladen.

Der Quelloperand kann im Stackelement ST(i) oder als REAL-Operand im Speicher stehen.

Die Speicheroperanden SHORT REAL, LONG REAL und TEMPORARY REAL sind moeglich.
SHORT- und LONG REAL-Operanden werden automatisch in das TEMPORARY REAL-Format konvertiert.

Befehlsablauf:

```
| TEMPORARY REAL<--ST(i)   |     | push stack (ST<--ST-1) |
| push stack (ST<--ST-1)   | o.  | ST<--mem-op            |
| ST<--TEMPORARY REAL      |
```

Kodierung:

```
 ST<--ST(i)
|_11011001_|_11000(i)_|

 ST<--REAL
|_11011m01_|_mod000r/m_|_disp.low_|_disp.high_|
 m=0 SHORT REAL, m=1 LONG REAL

 ST<--TEMPORARY REAL
|_11011011_|_mod101r/m_|_disp.low_|_disp.high_|
```

Beispiele:

```
FLD  ST(5)
FLD  WURZEL
```

7.1.2. Speichern REAL

Befehl:	FST	**store REAL**
Schreibweise:	FST ziel	

Exception-Flags: IE, OE, UE, PE

Operandenkombinationen:

Typ	ziel
1	ST(i)
2	mem-op

Befehlswirkung:

Ein Real-Operand wird vom Stacktop in ein anderes Stackelement ST(i) oder in den Speicher transportiert.
Als Speicheroperanden sind SHORT REAL und LONG REAL moeglich.

Der Signifikant wird in beiden Faellen in Uebereinstimmung mit dem RC-Feld im Kontrollwort gerundet und an das Speicher-Zielformat angepasst.

Der Exponent wird ebenfalls in die darstellbare Laenge und Bias des Zielformates konvertiert.

Der Signifikant wird nicht gerundet, wenn der Stacktop als SPECIAL gekennzeichnet ist. Er erhaelt dann ein oo, ein NAN oder ein DENORMAL.
In diesem Fall werden die letzten Signifikanten-Bits im Stacktop zur Anpassung an das Zielformat geloescht.

Der Exponent wird in gleicher Weise behandelt. Dadurch bleiben die Zahlen, die als oo oder NAN (Exponent entspricht dem Bitstring 11..11) bzw. als DENORMAL (Exponent entspricht dem Bitstring 00..00) identifiziert wurden, erhalten.

Der Stackpointer wird nicht veraendert.

Befehlsablauf: ST(i)<--ST oder mem-op<--ST

Kodierung:

```
 ST(i)<--ST
| 11011101 | 11010(i) |

 REAL<--ST
| 11011m01 | mod010r/m | disp.low | disp.high |
 m=0 SHORT REAL, m=1 LONG REAL
```

Beispiele:

FST ST(6)
FST ERGEBNIS

7.1.3. Speichern REAL und pop Stack

Befehl:	**FSTP**	**store REAL and pop**
Schreibweise:	**FSTP ziel**	

Exception-Flags: IE, OE, UE, PE

Operandenkombinationen:

Typ	ziel
1	ST(i)
2	mem-op

Befehlswirkung:

Ein REAL-Operand wird vom Stacktop in ein anderes Stackelement ST(i) oder in den Speicher transportiert.

Als Speicheroperanden sind SHORT REAL, LONG REAL und TEMPORARY REAL moeglich.
Sind die Speicheroperanden SHORT REALs oder LONG REALs, werden ihre Signifikanten in Uebereinstimmung mit dem RC-Feld im Kontrollwort gerundet und damit an das Zielformat angepasst.

Der Exponent wird ebenfalls in die darstellbare Laenge und Bias des Zielformates konvertiert.

Nach der Operation wird der Stackpointer inkrementiert.

```
Befehlsablauf: | ST(i)<--ST                 | o. | mem-op<--ST                |
               |_pop stack (ST<--ST+1)_|       |_pop stack (ST<--ST+1)_|
```

Kodierung:

```
 ST(i)<--ST___________
|_11011101_|_11011(i)_|

 REAL<--ST______________________________________
|_11011m01_|_mod011r/m_|_disp.low_|_disp.high_|
 m=0 SHORT REAL, m=1 LONG REAL

 TEMPORARY_REAL<--ST____________________________
|_11011011_|_mod111r/m_|_disp.low_|_disp.high_|
```

Beispiele:

```
     FSTP  ST(3)
     FSTP  KONST
```

7.1.4. Vertauschen Stackelement mit Stacktop

Befehl:	**FXCH**	**exchange registers**
Schreibweise:	**FXCH ziel** **FXCH**	

Exception-Flags: IE

Operandenkombinationen:

Typ	ziel
1	ST(i)
2	kein Operand

Befehlswirkung:

Das Stacktopregister ST wird mit dem i-ten Stackregister nach dem Stacktop =ST(i) vertauscht.

Wird im Befehl kein Zielregister angegeben, erfolgt ein Vertauschen des Stacktopregisters ST mit dem ersten Stackregister nach dem Stacktop =ST(1).

Der Stackpointer bleibt unveraendert.

Befehlsablauf: TEMPORARY REAL<--ST(i)
ST(i)<--ST
ST<--TEMPORARY REAL

Kodierung:

11011001	11001(i)

Beispiele:

```
FXCH  ST(3)
FXCH
```

7.1.5. INTEGER laden

Befehl:	FILD	INTEGER load
Schreibweise:	FILD quelle	

Exception-Flags: IE

Operandenkombinationen:

Typ	quelle
1	mem-op

Befehlswirkung:

Ein INTEGER-Speicheroperand wird in das Stacktopregister geladen.

Zuerst wird der Stackpointer dekrementiert, danach wird der INTEGER-Operand in den neuen Stacktop geladen.

Formate der zu verwendenden Quelloperanden:

- WORD-INTEGER (2 Bytes)
- SHORT-INTEGER (4 Bytes)
- LONG-INTEGER (8 Bytes)

Befehlsablauf: push stack (ST<--ST-1)
ST<--mem-op

Kodierung:

```
 ST<--INTEGER_________________________________
 _11011m11_|_mod000r/m_|_disp.low_|_disp.high_|
 m=0 SHORT, m=1 WORD INTEGER

 ST<--LONG_INTEGER____________________________
|_11011111_|_mod101r/m_|_disp.low_|_disp.high_|
```

Beispiel:

FILD WERT

7.1.6. INTEGER speichern

Befehl:	FIST	**INTEGER store**
Schreibweise:	FIST ziel	

Exception-Flags: IE, PE

Operandenkombinationen:

Typ	ziel
1	mem-op

Befehlswirkung:

INTEGER wird vom Stacktop in den Speicher transportiert.

Der Stackpointer bleibt unbeeinflusst.

Der Befehl rundet zunaechst den Inhalt des Stacktopregisters in Uebereinstimmung mit dem RC-Feld im Kontrollwort nach INTEGER.

Das Ergebnis wird anschliessend in den Speicher transportiert. Im Zielspeicher duerfen Variable vom Typ WORD- und SHORT-INTEGER gespeichert werden.

Befehlsablauf: mem-op<--ST

Kodierung:

```
---------------------------------------------------------
|_11011m11_|_mod010r/m_|_disp.low_|_disp.high_|
 m=0 SHORT, m=1 WORD INTEGER
```

Beispiel:

FIST RECH1

7.1.7. INTEGER speichern und pop Stack

Befehl:	FISTP	**INTEGER store and pop**
Schreibweise:	FISTP ziel	

Exception-Flags: IE, PE

Operandenkombinationen:

Typ	ziel
1	mem-op

Befehlswirkung:

Dieser Befehl transportiert einen INTEGER-Operanden vom Stacktop (ST) in den Speicher.

Zuerst rundet der Befehl den Inhalt des Stacktopregisters in Uebereinstimmung mit dem RC-Feld im Kontrollwort nach INTEGER.

Das Ergebnis wird anschliessend in den Speicher transportiert.

Im Zielspeicher duerfen Variable vom Typ WORD-, SHORT- und LONG-INTEGER gespeichert werden.

Nach der Operation wird der Stackpointer dekrementiert.

Befehlsablauf: mem-op<--ST
pop stack (ST<--ST+1)

Kodierung:

```
 INTEGER<--ST_________________________________
|_11011m11_|_mod011r/m_|_disp.low_|_disp.high_|
 m=0 SHORT, m=1 WORD INTEGER
```

```
LONG_INTEGER<--ST_____________________________
|_11011111_|_mod111r/m_|_disp.low_|_disp.high_|
```

Beispiel:

    FISTP RECH2

7.1.8. PACKED DECIMAL in Stacktop laden

Befehl:	**FBLD**	**BCD load**
Schreibweise:	**FBLD quelle**	

Exception-Flags: IE

Operandenkombinationen:

Typ	quelle
1	mem-op

Befehlswirkung:

Dieser Befehl transportiert einen PACKED DECIMAL-Operanden an den Stacktop.

Zuerst wird der Quelloperand, der im PACKED DECIMAL-Format (BCD, 10 Bytes) im Speicher steht, in das entsprechende TEMPORARY REAL-Format gewandelt.

Der aktuelle Stackpointer wird anschliessend dekrementiert und danach die TEMPORARY REAL-Zahl in den neuen Stacktop geladen.

Beachte:

- PACKED DECIMAL-Zahlen des Quelloperanden muessen im Bereich von 0 - 9H liegen.
- Der Befehl prueft die ungueltigen Zahlen AH - FH nicht.
- Er liefert dann ein undefiniertes Ergebnis (INDEFINITE).

Befehlsablauf: push stack (ST<--ST-1)
                  ST<--mem-op

Kodierung:

```
_____________________________________________
|_11011111_|_mod100r/m_|_disp.low_|_disp.high_|
```

Beispiel:

FBLD RECH

7.1.9. Speichern PACKED DECIMAL und pop Stack

Befehl:	**FBSTP**	**BCD store and pop**
Schreibweise:	**FBSTP ziel**	

Exception-Flags: IE

Operandenkombinationen:

Typ	ziel
1	mem-op

Befehlswirkung:

Der Befehl transportiert einen PACKED DECIMAL-Operanden vom Stacktop (ST) in den Speicher.

Zuerst wird dabei der Inhalt des Stacktop, z.B. ST = Stackregister R3, in ein PACKED DECIMAL INTEGER-Format gewandelt.

Danach wird das Ergebnis im Speicher abgelegt / Bytes) und anschliessend der Stackpointer inkrementiert.

Beachte:

- FBTSP liefert von einem nichtganzzahligen Operanden den aufgerundeten INTEGER-Wert.
- Dabei wird 0,5 zum Operanden addiert und anschliessend die Nachkommastellen abgeschnitten.

Befehlsablauf: mem-op<--ST
pop stack (ST<--ST+1)

Kodierung:

| 11011111 | mod110r/m | disp.low | disp.high |

Beispiel:

FBSTP [BX].STAT

7.2. Arithmetische Befehle

Uebersicht:

	Addition
FADD	add REAL
FADDP	add REAL and pop
FIADD	INTEGER add
	Subtraktion
FSUB	subtract REAL
FSUBP	subtract REAL and pop
FISUB	INTEGER subtract
FSUBR	subtract REAL reversed
FSUBRP	subtract REAL reversed and pop
FISUBR	INTEGER subtract reversed
	Multiplikation
FMUL	multiply REAL
FMULP	multiply REAL and pop
FIMUL	INTEGER multiply
	Division
FDIV	divide REAL
FDIVP	divide REAL and pop
FIDIV	INTEGER divide
FDIVR	divide REAL reversed
FDIVRP	divide REAL reversed and pop
FIDIVR	INTEGER divide reversed
	weitere Operationen
FSQRT	square root
FSCALE	scale
FPREM	partial remainder
FRNDINT	round to INTEGER
FXTRACT	extract exponent and significand
FABS	absolute value
FCHS	change sign

Neben den arithmetischen Basisoperationen (Addition, Subtraktion, Multiplikation, Division) koennen zusaetzliche Operationen wie z.B. Quadratwurzel und Absolutwert ausgefuehrt werden.

Als Operanden sind die WM87-Datentypen wie LONG REAL, SHORT REAL, SHORT INTEGER oder WORD INTEGER zu verwenden.

Die Operanden sind in Stackelementen oder im Datenspeicher abgelegt.

Das Ergebnis der Operation wird in einem Stackelement gespeichert.

7.2.1. REAL addieren

Befehl:	**FADD**	**add REAL**
Schreibweise:	**FADD ziel,quelle** **FADD quelle**	

Exception-Flags: IE, DE, OE, UE, PE

Operandenkombinationen:

Typ	ziel	quelle
1	ST(i)	ST
2	ST	ST(i)
3		mem-op

Befehlswirkung:

FADD addiert einen Quelloperanden zu einem Zieloperanden und ueberschreibt den Zieloperanden mit dem Additionsergebnis.

Der Quelloperand steht entweder im Stacktop, in einem Stackelement oder als SHORT REAL-(4 Bytes) bzw. LONG REAL-Operand (8 Bytes) im Speicher.
Steht der Quelloperand im Stacktop, ist der Zieloperand in einem Stackelement gespeichert.
Befindet sich der Quelloperand in einem Stackelement oder im Speicher, ist der Zieloperand im Stacktop gespeichert.

Der Stackpointer bleibt unveraendert.

Befehlsablauf: ziel<--ziel+quelle

```
ST(i)<--ST(i)+ST   oder
ST<--ST+ST(i)      oder
ST<--ST+mem-op
```

Kodierung:

```
 ST(i)+ST_____________
|_11011d00_|_11000(i)_|

 ST+REAL___________________________________
|_11011m00_|_mod000r/m_|_disp.low_|_disp.high_|
 m=0 SHORT REAL, m=1 LONG REAL
```

Beipiele:

```
FADD  ST(3),ST
FADD  ST,ST(2)
FADD  BETRAG
```

7.2.2. REAL addieren und pop stack

Befehl:	**FADDP**	**add REAL and pop**
Schreibweise:	**FADDP ziel,quelle** **FADD**	

Exception-Flags: IE, DE, OE, UE, PE

Operandenkombinationen:

Typ	ziel	quelle
1	ST(i)	ST

Befehlswirkung:

Mit diesem Befehl wird der Quelloperand im Stacktop zum i-ten Stackregister nach Stacktop addiert.
Das Ergebnis ueberschreibt den Zieloperanden.

Nach der Operation wird der Stackpointer inkrementiert.

Wird zum Befehl FADD kein Operand angegeben, wird ST zu ST(1) addiert.

Befehlsablauf: ziel<--ziel+quelle

```
|¯ST(i)<--ST(i)+ST        ¯|
|_pop stack (ST<--ST+1)_|      oder

|¯ST(1)<--ST(1)+ST        ¯|
|_pop stack (ST<--ST+1)_|
```

Kodierung:

```
|_11011110_|_11000(i)_|
```

Beispiele:

```
FADDP  ST(3),ST
FADD
```

7.2.3. INTEGER addieren

Befehl:	**FIADD**	**INTEGER add**
Schreibweise:	**FIADD quelle**	

Exception-Flags: IE, DE, OE, PE

Operandenkombinationen:

Typ	quelle
1	mem-op

Befehlswirkung:

Dieser Befehl addiert einen Speicher-INTEGER-Operanden mit dem Stacktop. Die Summe ueberschreibt das Stacktopregister.

Als Speicheroperanden sind WORD INTEGER und SHORT INTEGER moeglich.

Der Stackpointer bleibt unveraendert.

Befehlsablauf: ziel<--ziel+quelle

ST<--ST+mem-op

Kodierung:

```
| 11011m10 | mod000r/m | disp.low | disp.high |
 m=0 SHORT, m=1 WORD INTEGER
```

Beispiel:

FIADD ZAEHLER1

7.2.4. REAL subtrahieren

Befehl:	**FSUB**	subtract REAL
Schreibweise:	**FSUB ziel,quelle** **FSUB quelle**	

Exception-Flags: IE, DE, OE, UE, PE

Operandenkombinationen:

Typ	ziel	quelle
1	ST(i)	ST
2	ST	ST(i)
3		mem-op

Befehlswirkung:

Der Befehl subtrahiert einen Quelloperanden von einem Zieloperanden und ueberschreibt den Zieloperanden mit dem Subtraktionsergebnis.

Der Quelloperand steht im Stacktop, in einem Stackelement oder als SHORT REAL- bzw. LONG REAL-Operand im Speicher.
Steht der Quelloperand im Stacktop ST, dann ist der Zieloperand in einem der Stackelemente ST(i) gespeichert.
Steht der Quelloperand in einem Stackelement oder im Speicher, ist der Zieloperand im Stacktop abgelegt.

Der Stackpointer bleibt unveraendert.

Befehlsablauf: ziel<--ziel-quelle

```
ST(i)<--ST(i)-ST        oder
ST<--ST-ST(i)           oder
ST<--ST-mem-op
```

Kodierung:

```
 ST(i)-ST, ST-ST(i)
|_11011d00_|_11100(i)_|

 ST-REAL
|_11011m00_|_mod100r/m_|_disp.low_|_disp.high_|
 m=0 SHORT REAL, m=1 LONG REAL
```

Beispiele:

```
FSUB  ST(3),ST
FSUB  ST,ST(4)
FSUB  FELD1
```

7.2.5. REAL subtrahieren und pop Stack

Befehl:	**FSUBP**	**subtract REAL and pop**
Schreibweise:	**FSUBP ziel,quelle** **FSUB**	

Exception-Flags: IE, DE, OE, UE, PE

Operandenkombinationen:

Typ	ziel	quelle
1	ST(i)	ST

Befehlswirkung:

Der Quelloperand wird vom i-ten Stackregister nach Stacktop subtrahiert. Das Ergebnis ueberschreibt den Zieloperanden.

Nach der Operation wird der Stackpointer inkrementiert.

Wird zum Befehl FSUB kein Operand angegeben, erfolgt eine Subtraktion des Quelloperanden ST vom Zieloperanden ST(1).

Befehlsablauf: ziel<--ziel-quelle

```
|¯ST(i)<--ST(i)-ST       ¯|
|_pop stack (ST<--ST+1)_|          oder

|¯ST(1)<--ST(1)-ST       ¯|
|_pop stack (ST<--ST+1)_|
```

Kodierung:

```
 ----------------------
|_11011110_|_11100(i)_|
```

Beispiele:

```
FSUBP ST(4),ST
FSUB
```

7.2.6. INTEGER subtrahieren

Befehl:	**FISUB**	**INTEGER subtract**
Schreibweise:	**FISUB quelle**	

Exception-Flags: IE, DE, OE, PE

Operandenkombinationen:

Typ	quelle
1	mem-op

Befehlswirkung:

FISUB subtrahiert einen Speicher-INTEGER-Operanden vom Stacktop. Die Differenz ueberschreibt das Stacktopregister.

Als Speicheroperanden sind WORD INTEGER und SHORT INTEGER moeglich.

Der Stackpointer bleibt unveraendert.

Befehlsablauf: ziel<--ziel-quelle

ST<--ST-mem-op

Kodierung:

```
|_11011m10_|_mod100r/m_|_disp.low_|_disp.high_|
```

Beispiel:

FISUB ZAEHLER2

7.2.7. INTEGER umgekehrt subtrahieren

Befehl:	**FISUBR**	**INTEGER subtract reversed**
Schreibweise:	**FISUBR quelle**	

Exception-Flags: IE, DE, OE, PE

Operandenkombinationen:

Typ	quelle
1	mem-op

Befehlswirkung:

Dieser Befehl subtrahiert das Stacktopregister von einem Speicher-INTEGER-Operanden.

Die Differenz ueberschreibt das Stacktopregister.

Als Speicheroperanden koennen WORD INTEGER und SHORT INTEGER verwenden werden.

Der Stackpointer veraendert sich nicht.

Befehlsablauf: ziel<--quelle-ziel

ST<--mem-op-ST

Kodierung:

```
|_11011m10_|_mod101r/m_|_disp.low_|_disp.high_|
 m=0 SHORT, m=1 WORD INTEGER
```

Beispiel:

FISUBR KONST

7.2.8. REAL umgekehrt subtrahieren

Befehl:	**FSUBR**	**subtract REAL reversed**
Schreibweise:	**FSUBR ziel,quelle** **FSUBR quelle**	

Exception-Flags: IE, DE, OE, UE, PE

Operandenkombinationen:

Typ	ziel	quelle
1	ST	ST(i)
2	ST(i)	ST
3		mem-op

Befehlswirkung:

Bei dem Befehl FSUBR wird der Zieloperand vom Quelloperanden subtrahiert. Die Differenz ueberschreibt den Zieloperanden.

Als Quelloperanden sind der Inhalt des Stacktop, der Inhalt eines Stackregisters oder ein SHORT- bzw. LONG REAL-Speicheroperand moeglich.
Befindet sich der Quelloperand im Stacktop, ist der Zieloperand in einem der Stackregister abgelegt.
Steht der Quelloperand in einem Stackregister oder ist er ein Speicheroperand, ist der Zieloperand im Stacktop gespeichert.

Der Stackpointer bleibt unbeeinflusst.

Befehlsablauf: ziel<--quelle-ziel

```
ST<--ST(i)-ST        oder
ST(i)<--ST-ST(i)     oder
ST<--mem-op-ST
```

Kodierung:

```
 ST(i)-ST, ST-ST(i)
| 11011d00 | 11101(i) |

 REAL-ST
| 11011m00 | mod101r/m | disp.low | disp.high |
 m=0 SHORT REAL, m=1 LONG REAL
```

Beispiele:

```
FSUBR ST,ST(4)
FSUBR ST(4),ST
FSUBR TAB
```

7.2.9. REAL umgekehrt subtrahieren und pop Stack

Befehl:	FSUBRP	subtract REAL reversed and pop
Schreibweise:	FSUBRP ziel,quelle FSUBR	

Exception-Flags: IE, DE, OE, UE, PE

Operandenkombinationen:

Typ	ziel	quelle
1	ST(i)	ST

Befehlswirkung:

Vom Quelloperanden wird das i-te Stackregister nach Stacktop subtrahiert. Das Ergebnis ueberschreibt den Zieloperanden.

Nach der Operation wird der Stackpointer inkrementiert.

Wird zum Befehl FSUBR kein Operand angegeben, dann wird von ST das erste Stackregister nach dem Stacktop subtrahiert.

Befehlsablauf: ziel<--quelle-ziel

```
| ST(i)<--ST-ST(i)          |
| pop stack (ST<--ST+1)     |   oder

| ST(1)<--ST-ST(1)          |
| pop stack (ST<--ST+1)     |
```

Kodierung:

| 11011110 | 11101(i) |

Beispiele:

```
FSUBRP ST(4),ST
FSUBR
```

7.2.10. REAL multiplizieren

Befehl:	FMUL	multiply REAL
Schreibweise:	FMUL ziel,quelle FMUL quelle	

Exception-Flags: IE, DE, OE, UE, PE

Operandenkombinationen:

Typ	ziel	quelle
1	ST	ST(i)
2	ST(i)	ST
3		mem-op

Befehlswirkung:

Der Zieloperand wird mit dem Quelloperanden multipliziert. Das Produkt ueberschreibt den Zieloperanden.

Der Quelloperand kann im Stacktop, in einem Stackelement oder als SHORT REAL- bzw. LONG REAL-Operand im Speicher stehen.

Wenn der Quelloperand im Stacktop steht, ist der Zieloperand in einem der Stackelemente gespeichert.

Steht der Quelloperand in einem Stackelement oder im Speicher, dann wird der Zieloperand im Stacktop gespeichert.

Der Stackpointer bleibt unveraendert.

Befehlsablauf: ziel<--ziel*quelle

```
ST<--ST*ST(i)        oder
ST(i)<--ST(i)*ST     oder
ST<--ST*mem-op
```

Kodierung:

```
ST*ST(i)
I 11011d00 I 11001(i) I

ST*REAL
I 11011m00 I mod001r/m I disp.low I disp.high I
m=0 SHORT REAL, m=1 LONG REAL
```

Beispiele:

```
FMUL  ST,ST(4)
FMUL  ST(5),ST
FMUL  KOSTEN
```

7.2.11. REAL multiplizieren und pop Stack

Befehl:	FMULP	**multiply REAL and pop**
Schreibweise:	**FMULP ziel,quelle** FMUL	

Exception-Flags: IE, DE, OE, UE, PE

Operandenkombinationen:

Typ	ziel	quelle
1	ST(i)	ST

Befehlswirkung:

Das i-te Stackregister nach dem Stacktop wird mit dem Stacktop multipliziert. Das Produkt ueberschreibt den Zieloperanden.

Nach der Multiplikation wird der Stackpointer inkrementiert.

Wird beim Befehl FMUL kein Operand angegeben, dann wird der Zieloperand ST(1) mit dem Quelloperanden ST multipliziert.

Befehlsablauf: ziel<--ziel*quelle

```
| ST(i)<--ST(i)*ST        |
|_pop stack (ST<--ST+1)_|
                                    oder
| ST(1)<--ST(1)<--ST      |
|_pop stack (ST<--ST+1)_|
```

Kodierung:

11011110	11001(i)

Beispiel:

```
FMULP  ST(4),ST
FMUL
```

7.2.12. INTEGER Multiplizieren

Befehl:	FIMUL	INTEGER multiply
Schreibweise:	FIMUL quelle	

Exception-Flags: IE, DE, OE, PE

Operandenkombinationen:

Typ	quelle
1	mem-op

Befehlswirkung:

Dieser Befehl multipliziert einen Speicher-INTEGER-Operanden mit dem Stacktop.

Das Produkt ueberschreibt das Stacktopregister.

Es sind WORD INTEGER- und SHORT INTEGER-Speicheroperanden moeglich.
Der Stackpointer bleibt unveraendert.

Befehlsablauf: ziel<--ziel*quelle

ST<--ST*mem-op

Kodierung:

11011m10	mod001r/m	disp.low	disp.high

Beispiel:

FIMUL KONST

7.2.13 REAL dividieren

Befehl:	FDIV	divide REAL
Schreibweise:	FDIV ziel,quelle FDIV quelle	

Exception-Flags: IE, DE, ZE, OE, UE, PE

Operandenkombinationen:

Typ	ziel	quelle
1	ST	ST(i)
2	ST(i)	ST
3		mem-op

Befehlswirkung:

Dieser Befehl dividiert den Zieloperanden durch den Quelloperanden. Der Quotient ueberschreibt den Zieloperanden.

Der Quelloperand kann im Stacktop, in einem Stackelement oder als SHORT REAL- bzw. LONG REAL-Operand im Speicher stehen.
Wenn der Quelloperand im Stacktop steht, ist der Zieloperand in einem der Stackelemente gespeichert.
Steht der Quelloperand in einem Stackelement oder im Speicher, dann wird der Zieloperand im Stacktop gespeichert.

Der Stackpointer bleibt unveraendert.

Befehlsablauf: ziel<--ziel/quelle

```
ST(i)<--ST(i)/ST      oder
ST<--ST/ST(i)         oder
ST<--ST/mem-op
```

Kodierung:

```
 ST(i)/ST, ST/ST(i)
|_11011d00_|_11110(i)_|

 ST/REAL
|_11011m00_|_mod110r/m_|_disp.low_|_disp.high_|
 m=0 SHORT REAL, m=1 LONG REAL
```

Beispiele:

```
FDIV  ST,ST(2)
FDIV  ST(3),ST
FDIV  ARCUS
```

7.2.14. REAL dividieren und pop Stack

Befehl:	FDIVP	divide REAL and pop
Schreibweise:	FDIVP ziel,quelle FDIV	

Exception-Flags: IE, DE, ZE, OE, UE, PE

Operandenkombinationen:

Typ	ziel	quelle
1	ST(i)	ST

Befehlswirkung:

Das i-te Stackregister nach Stacktop wird durch das Stacktopregister dividiert. Der Quotient ueberschreibt das Zielregister.

Nach der Division wird der Stackpointer inkrementiert.

Wird beim Befehl FDIV kein Operand angegeben, dann wird der Zieloperand ST(1) durch den Quelloperanden ST dividiert.

Befehlsablauf: ziel<--ziel/quelle

ST(i)<--ST(i)/ST
pop stack (ST<--ST+1)

Kodierung:

```
 _____________________
|_11011110_|_11110(i)_|
```

__Beispiele:__

FDIVP ST(2),ST
FDIV

7.2.15 INTEGER dividieren

Befehl:	FIDIV	INTEGER divide
Schreibweise:	FIDIV ziel	

Exception-Flags: IE, DE, ZE, OE, UE, PE

Operandenkombinationen:

Typ	ziel
1	mem-op

Befehlswirkung:

Bei diesem Befehl wird der Divident im Stacktopregister durch den Divisor, der als Quelloperand im Speicher steht, dividiert. Der Quelloperand kann WORD INTEGER oder SHORT INTEGER sein.

Der Quotient ueberschreibt das Stacktopregister.

Der Stackpointer bleibt unveraendert.

Befehlsablauf: ziel<--ziel/quelle

ST<--ST/mem-op

Kodierung:

```
|_11011m10_|_mod110r/m_|_disp.low_|_disp.high_|
```

Beispiel:

FIDIV TEILERGEB

7.2.16 INTEGER umgekehrt dividieren

Befehl:	**FIDIVR**	**INTEGER divide reversed**
Schreibweise:	**FIDIVR ziel**	

Exception-Flags: IE, DE, ZE, OE, UE, PE

Operandenkombinationen:

Typ	ziel
1	mem-op

Befehlswirkung:

Der Befehl FIDIVR dividiert eine WORD- oder SHORT-INTEGER-Zahl im Speicher durch den Divisor im Stacktopregister.

Der Quotient ueberschreibt das Stacktopregister.

Der Stackpointer bleibt unbeeinflusst.

Befehlsablauf: ziel<--quelle/ziel

ST<--mem-op/ST

Kodierung:

```
|_11011m10_|_mod111r/m_|_disp.low_|_disp.high_|
```

Beispiel:

FDIVR TABWERT

7.2.17. REAL umgekehrt dividieren

Befehl:	FDIVR	divide REAL reversed
Schreibweise:	FDIVR ziel,quelle FDIVR quelle	

Exception-Flags: IE, DE, ZE, OE, UE, PE

Operandenkombinationen:

Typ	ziel	quelle
1	ST	ST(i)
2	ST(i)	ST
3		mem-op

Befehlswirkung:

Dieser Befehl dividiert den Quelloperenden durch den Zieloperanden. Der Quotient ueberschreibt den Zieloperanden.
Der Quelloperand kann im Stacktop, in einem Stackelement oder als SHORT REAL oder LONG REAL im Speicher stehen.
Steht der Quelloperand im Stacktop, befindet sich der Zieloperand in einem der Stackelemente.

Ist der Quelloperand in einem Stackelement oder im Speicher abgelegt, dann befindet sich der Zieloperand im Stacktop.

Der Stackpointer bleibt unveraendert.

Befehlsablauf: ziel<--quelle/ziel

```
ST<--ST(i)/ST        oder
ST(i)<--ST/ST(i)     oder
ST<--mem-op/ST
```

Kodierung:

```
 ST(i)/ST, ST/ST(i)
| 11011d00 | 11111(i) |

 REAL/ST
| 11011m00 | mod111r/m | disp.low | disp.high |
 m=0 SHORT REAL, m=1 LONG REAL
```

Beispiele:

```
FDIVR  ST,ST(3)
FDIVR  ST(4),ST
FDIVR  DS:TAB
```

7.2.18 REAL umgekehrt dividieren und pop Stack

Befehl:	FDIVRP	divide REAL reversed and pop
Schreibweise:	FDIVRP ziel,quelle FDIVR	

Exception-Flags: IE, DE, ZE, OE, UE, PE

Operandenkombinationen:

Typ	ziel	quelle
1	ST(i)	ST

Befehlswirkung:

Das Stacktopregister, der Quelloperand, wird durch das i-te Stackregister nach Stacktop dividiert. Der Quotient ueberschreibt das Zielregister.

Nach der Operation wird der Stackpointer inkrementiert.

Wird beim Befehl FDIVR kein Operand angegeben, dann wird das Stacktopregister ST durch das erste Register nach Stacktop ST(1) dividiert.

Befehlsablauf: ziel<--quelle/ziel

```
| ST(i)<--ST/ST(i)          |
| pop stack (ST<--ST+1)     |   oder

| ST(1)<--ST/ST(1)          |
| pop stack (ST<--ST+1)     |
```

Kodierung:

```
| 11011110 | 11111(i) |
```

Beispiele:

```
FDIVRP  ST(5),ST
FDIVR
```

7.2.19. Quadratwurzel

Befehl:	FSQRT	**Square root**
Schreibweise:	FSQRT	

Exception-Flags: IE, DE, PE

Befehlswirkung:

Die Quadratwurzel wird vom Stacktop ST berechnet und wieder in ST abgespeichert.
Das Argument muss im Bereich -0 <= ST <= +oo liegen.

Der Stackpointer bleibt unveraendert.

Befehlsablauf: ST<-- $\sqrt{ST}$

Kodierung:

| 11011001 | 11111010 |

Beispiel:

FSQRT

7.2.20 Funktion z=Y*2^X berechnen

Befehl:	FSCALE	**scale**
Schreibweise:	FSCALE	

Exception-Flags: IE, OE, UE

Befehlswirkung:

Der Operand Y steht im Stacktop ST und der Exponent im ersten Stackregister nach Stacktop ST(1). Nach Ausfuehren des Befehles wird ST ueberschrieben.
Der Stackpointer bleibt unveraendert.

Der Befehl FSCALE interpretiert die Groesse in ST(1) als eine Groesse vom Type INTEGER.
Die Groesse muss im Bereich -2^{15} <= ST(1) <= $+2^{15}$ liegen.

Der Exponent (Scale-Faktor) sollte von einem WORD INTEGER-Feld geladen werden.

Fuer den Groessenbereich des Scale-Faktors ist folgendes zu beachten:

- Ist der SCALE-Faktor eine REAL-Zahl, die im Bereich liegt, und deren Betrag > 1 ist, werden die Nachkommastellen abgeschnitten.
- $-2^{15} <= ST(1) < +2^{15}$

```
         -----
           |
           v
      |REAL-Zahl| > 1
```

- Liegt der Scale-Faktor ausserhalb des Bereiches oder liegt sein Betrag im Bereich von Ø bis 1, ergibt sich ein undefiniertes Ergebnis.

- Es erfolgt eine Meldung in den EXCEPTION-Flags IE, UE, OE.

$ST(1) < -2^{15}$

oder

$ST(1) >= +2^{15}$

oder

$Ø < |ST(1)| < 1$

Befehlsablauf: $ST <-- ST * 2^{ST(1)}$

Kodierung:

```
 ---------------------
|_11011001_|_11111101_|
```

Anwendung:

Soll eine Potenz, deren Exponent x>Ø.5 ist, berechnet werden, ist es notwendig, die Potenz in berechenbare Teilpotenzen zu zerlegen.

<u>Beispiel:</u>

```
2^x = 2^26.8496

    = 2^26  *  2^Ø.5  *  2^Ø.3496
      ---     -----     -------
       |        |          |
       |        |          v
       |        |
       |        v        Mit F2XM1 berechenbar.
       |
       |      ≙ \/2
       v      Mit FSQRT berechenbar.
      ≙ +1 * 2^26
          |    |
          |    v
          |
          v   ST(1)

         ST
      Mit FSCALE berechenbar, den ganzzahligen Anteil von
      26.8496 = 26 erhaelt man durch FRNDINT.
```

7.2.21. Modulo-Division

Befehl:	FPREM	partial remainder
Schreibweise:	FPREM	

Exception-Flags: IE, DE, UE

Befehlswirkung:

Mit FPREM wird eine Modulo-Division zwischen dem Inhalt des Stacktop ST (Divident) und dem Inhalt des Stackregisters nach Stacktop ST(1) durchgefuehrt.
Das Ergebnis ist der Rest (Remainder), der den Dividenten ueberschreibt.

ST<--ST MOD ST(1)

Das Vorzeichen des Restes entspricht dem Vorzeichen des Dividenten.
Der Befehl liefert ein exaktes Ergebnis, es wird keine PRECISION-Exception ausgegeben. Bei der Modulo-Division subtrahiert FPREM den Modulus ST(1) (Divisor) vom Dividenten ST.

ST<--ST-ST(1)

Beim einmaligen Ausfuehren des Befehles darf die Differenz zwischen dem Dividenten und dem Modulus bis zu 2^{64} betragen.
Liefert FPREM einen Rest, der kleiner als der Modulus ist, ist die Funktion beendet.

```
 ST  <--      ST    -   ST(1)
-----        -----     -----
  |            |         |
  |            |         v
  |            |        Modulus
  |            v
  v
             Divident
  Rest< Modulus
  ST  < ST(1)
```

In diesem Fall wird das CONDITION-Bit C2 im Statuswort geloescht (C2=0).
Wenn der Rest 0 ist, wird die Funktion ebenfalls beendet.

```
ST       <--  ST    -   ST(1)
-----        -----     -----
  |            |         |
  |            |         v
  |            |        Modulus
  |            v
  v
             Divident
 Rest = 0
 ST   = ST(1)
```

Hier sind der Divident und der Modulus identisch. Das CONDITION-Bit im Statuswort wird geloescht (C2=0).
Die Funktion ist nicht beendet, wenn FPREM einen Rest liefert, der groesser als der Modul ist. Das CONDITION-Bit C2 im Statuswort wird gesetzt.

```
 ST     <--      ST     -    ST(1)
-----           -----       -----
  |               |           |
  |               |           v
  |               |          Modulus
  |               v
  |
  v             Divident

   Rest > 0
   ST   > ST(1)
```

Das Ergebnis im Stacktop ist der **Partial Remainder** (Teilergebnis).
Der FPREM-Befehl ist in einer Programmschleife so oft zu wiederholen, bis der Rest < Modulus oder der Rest = 0ist.
Dabei wird der Partial Remainder in ST als Divident verwendet.

ST <-- REPEAT [ST - ST(1)]

Verfahren:
Nach dem Ausfuehren des FPREM-Befehles wird zuerst das Statuswort des WM87 in den Speicher geschrieben (Befehl: FSTSW mem-op).
Danach kann mit einem WM86-Maskenbefehl (AND mem,data) die Wertigkeit des CONDITION-Bits C2 geprueft werden:
Bei C2 = 1 ist FPREM zu wiederholen (REPEAT).
Bei C2 = 0 ist die Funktion beendet.

Der Stackpointer bleibt unveraendert.

Kodierung:

```
 ---------------------
| 11011001 | 11111000 |
 ---------------------
```

Anwendung:

Bei periodisch transzendentalen Funktionen ist es notwendig, die Argumente in einen Bereich zu bringen, der beim Verwenden entsprechender Befehle erlaubt ist.

Beispiel:

Zur Berechnung einer Tangensfunktion ist der Befehl FPTAN zu verwenden.
Da dieser Befehl nur Argumente im Bereich von $0 < \theta <= \frac{pi}{4}$ erlaubt, ist fuer groessere Argumente ($\theta < \frac{pi}{4}$) mit dem FPREM-Befehl das so zu reduzieren, dass es in den Bereich $0 < \theta <= \frac{pi}{4}$ zu liegen kommt.

7.2.22. Zu INTEGER-Zahl runden

Befehl:	FRNDINT	**round to INTEGER**
Schreibweise:	FRNDINT	

Exception-Flags: IE, PE

Befehlsablauf:

Eine REAL-Zahl, die im Stacktop gespeichert ist, wird in eine INTEGER-Zahl umgewandelt.
Nach Ausfuehren des Befehles steht die INTEGER-Zahl im Stacktop.

Das RC-Feld im Kontrollwort des NDP gibt die Art des Rundens an.

Der Stackpointer bleibt unveraendert.

Befehlsablauf: ST<--naechstgelegene INTEGER-Groesse von ST

Kodierung:

| 11011001 | 11111100 |

Beispiel:

Die REAL-Zahl 255.789 ist im Stacktopregister gespeichert. Das RC-Feld im Kontrollwort des WM87 soll wie folgt gesetzt sein:

| RC |
| 01 | ≙ Abrunden in Richtung -oo

oder

| RC |
| 11 | ≙ Abschneiden in Richtung 0

In beiden Faellen steht nach Ausfuehren des FRNDINT-Befehles im Stacktopregister die INTEGER-Zahl 255.

Ist das RC-Feld im Kontrollwort wie folgt gesetzt:

| RC |
| 10 | ≙ Aufrunden in Richtung +oo

oder

| RC |
| 11 | ≙ Runden in Richtung der naechstgelegenen INTEGER-Zahl

Dann steht in beiden Faellen nach Ausfuehren des FRNDINT-Befehles im Stacktopregister die INTEGER-Zahl 256.

7.2.23. Exponent und Signifikant trennen

Befehl:	FXTRACT	**extract exponent and significand**
Schreibweise:	FXTRACT	

Exception-Flags: IE

Befehlsablauf:

Dieser Befehl zerlegt einen Operanden im Stacktop in seinen Signifikanten und seinen Exponenten und stellt ihre Groesse in REAL dar.
Dabei werden zuerst der Exponent und der Signifikant in das TEMPORARY REAL-Format uebertragen, und anschliessend ueberschreibt der Exponent den Originaloperanden im Stacktop.

Der Stackpointer wird dekrementiert und der Signifikant in den neuen Stacktop geschrieben.

Nach dem Ausfuehren von FXTRACT enthaelt der neue Stacktop den Signifikanten als REAL-Zahl dargestellt (SIGN-, EXPONENT- und SIGNIFICAND-Feld).
Im SIGN-Feld steht das Vorzeichen des Operanden, im EXPONENT-Feld steht eine "wahre Null" in biased-Darstellung (16 383D = 3FFFH fuer TEMPORARY REAL) und im SIGNIFICAND-Feld ist der Signifikant gespeichert.

Der vorherige Stacktop, jetzt ST(1), enthaelt den wahren Exponenten (unbiased) des Originaloperanden, dargestellt als REAL-Zahl.

Beachte:

Ist der Originaloperand Null, liefert FXTRACT sowohl im Stacktop (ST) als auch im Stacktop + 1 (ST(1)) Null.
Beide Nullen haben das Vorzeichen des Originaloperanden.

Befehlsablauf:

```
T1<--Exponent (ST)
T2<--Signifikand (ST)
ST<--T1
push stack (ST<--ST-1)
ST<--T2
```

Kodierung:

11011001	11110100

Beispiele:

Der Stacktop soll eine Zahl enthalten, deren wahrer Exponent +4 ist. Das EXPONENT-Feld enthaelt somit:

E = e + 3FFFH = 4 + 3FFFH = 4003H (biased-Form)

Nach dem Ausfuehren des Befehles FXTRACT enthaelt ST(1), vorher Stacktop, +4.
Im SIGN-Feld steht ein positives Vorzeichen (Ø).
Das EXPONENT-Feld enthaelt 4ØØ1H (der wahre Exponent ist +2).
Im SIGNIFICAND-Feld steht 1‸ØØ...ØØB.

In ST(1) ist also die REAL-Zahl $1.Ø*2^{2}$ = 4 gespeichert.

Im zweiten Beispiel soll der Stacktop eine Zahl enthalten, deren wahrer Exponent -7 ist.
Im EXPONENTEN-Feld ist gespeichert:

E = e + 3FFFH = - 7 + 3FFFH = 3FF8H (biased-Form)

Nach Ausfuehren von FXTRACT enthaelt ST(1), vorher Stacktop, -7.
Im SIGN-Feld steht ein negatives Vorzeichen (1).
Das Exponent-Feld enthaelt 4ØØ1H (der wahre Exponent ist +2).
Das SIGNIFICAND-Feld enthaelt 1‸11ØØ...ØØB.

In ST(1) steht nun die REAL-Zahl $-1.11*2^{2}$ = -7.

Beachte:

Der Befehl FXTRACT kann in Verbindung mit FBSTP verwendet werden, TEMPORARY REAL-Zahlen in BCD-Formate z.B. fuer Anzeigen umzuwandeln.

7.2.24. Absolutwert bilden

Befehl:	**FABS**	**absolut value**
Schreibweise:	**FABS**	

Exception-Flags: IE

Befehlswirkung:

Der Befehl FABS bildet vom Inhalt des Stacktopregisters ST den Absolutwert, indem er das Vorzeichen positiv setzt (SIGN-Bit=Ø).

Der Stackpointer bleibt unveraendert.

Befehlsablauf: ST<--|ST|

Kodierung:

| 11Ø11ØØ1 | 111ØØØØ1 |

Beispiel:

FABS

7.2.25. Vorzeichen wechseln

Befehl:	FCHS	**change sign**
Schreibweise:	FCHS	

Exception-Flags: IE

Befehlsablauf:

FCHS komplementiert das Vorzeichen des Stacktopelementes.

Der Stackpointer bleibt unveraendert.

Befehlsablauf: ST<-- -ST

Kodierung:

| 11011001 | 11100000 |

Beispiel:

FCHS

7.3. Vergleichsbefehle

Uebersicht:

FCOM	compare REAL
FCOMP	compare REAL and pop
FCOMPP	compare REAL and pop twice
FICOM	INTEGER compare
FICOMP	INTEGER compare and pop
FTST	test
FXAM	examine

Die Vergleichsbefehle analysieren entweder den Inhalt des Stacktopelementes oder vergleichen das Stacktopelement mit anderen Operanden.

Das Vergleichsergebnis bzw. das Ergebnis der Analyse wird in beiden Faellen im CONDITION-Code des Statuswortes abgelegt.

7.3.1. REAL-Zahlen vergleichen

Befehl:	FCOM	compare REAL
Schreibweise:	FCOM quelle FCOM	

Exception-Flags: IE, DE
Condition-Code: CØ, C3

Operandenkombinationen:

Typ	quelle
1	ST(i)
2	mem-op

Befehlswirkung:

FCOM vergleicht einen REAL-Operanden im Stacktop mit einem REAL-Quelloperanden.
Der REAL-Quelloperand kann in einem beliebigen Stackregister, im Speicher als SHORT REAL- oder LONG REAL-Operand oder wenn kein Operand zum Befehl angegeben wird, im Stacktop +1 ST(1) stehen.
Der Vergleich wird durchgefuehrt, indem der Quelloperand vom Inhalt des Stacktopregisters subtrahiert wird.
Das Vergleichsergebnis wird in den CONDITION-Bits C3 und CØ im Statuswort hinterlegt.
Der Stackpointer bleibt unveraendert.

Befehlsablauf: ST -ST(1) oder
ST - ST(i) oder
ST - mem-op

Kodierung:

```
ST-ST(i)
11011000 | 11010(i) |

ST-REAL
| 11011m00 | mod010r/m | disp.low | disp.high |
m=Ø SHORT REAL, m=1 LONG REAL
```

Beispiele:

```
FCOM ST(4)
FCOM MAXIMUM
FCOM
```

CONDITION-Bits C3 und C0:

C3	C0	Bedeutung
0	0	ST > Quelloperand
0	1	ST < Quelloperand
1	0	ST = Quelloperand
1	1	ST ? Quelloperand

Beachte:

Bei Verwendung der **affinen** Zahlendarstellung werden vorzeichenbehaftete Nullen (+0, -0) durch den Vergleichsbefehl identisch behandelt, als waeren sie vorzeichenlos.
Vorzeichenlose Groessen oo bei Verwendung der **projektiven** Zahlendarstellung und die Spezialgroesse NAN (Not-A-Number) fuer Vergleichsoperationen koennen nicht verwendet werden.
Wird eine Vergleichsoperation mit oo oder NAN durchgefuehrt, dann erfolgt die Meldung im Statuswort C3 = 1, C0 = 1.

Vergleichsergebnis testen:

Nach dem Ausfuehren des Vergleichsbefehles kann mit folgender Prozedur das Statuswort des WM87 gespeichert und anschliessend zur Ueberpruefung des Ergebnisses in das WM86-Flagregister geladen werden:

```
FSTSW STAT_87               ;Statuswort speichern nach Aus-
                            ;fuehren von FCOM
FWAIT                       ;Warten bis Speichern beendet
MOV AH,BYTE PTR STAT_87+1   ;Lade STATUS-Byte in das AH-
                            ;Register des WM86
SAHF                        ;Lade STATUS-Byte in das WM86-
                            ;Flagregister
```

Im WM86-Flagregister stehen nach Ausfuehren des Befehles SAHF folgende WM87-Statuswortinformationen:

```
         WM86-Register
   7                             0
  | B |C3 |     ST    |C2 |C1 |C0 |        AH-Register
  |___|___|___|___|___|___|___|___|
    |   |___    |___    |__     |
    |       |______ |__   |     |
    |__________   |   |   |     |
                  v   v   v   v   v
 O   D   I   T   S   Z   A   P   C
|   |   |   |   |   |C3 |   |   |C0 |     Flagregister
|___|___|___|___|___|___|___|___|___|
```

Um das Vergleichsergbnis nach einem FCOM-Befehl zu untersuchen, koennen die bedingten Sprungbefehle des WM86 verwendet werden, die sich auf das
- C-Flag = CONDITION-Bit C0 und das
- Z-Flag = CONDITION-Bit C3 beziehen.

Anwendbare Sprungbefehle und ihre Bedeutung:

Befehl	Bedingung	Wirkung
JA disp	C I C0 Z I C3 0 und 0	Springe, wenn: ST>Quelloperand
JAE disp	C I C0 0	Springe, wenn: ST>=Quelloperand
JB disp	C I C0 1	Springe, wenn: ST<Quelloperand
JBE disp	C I C0 Z I C3 1 oder 1	Springe, wenn: ST<=Quelloperand
JE disp	Z I C3 1	Springe, wenn:
JNE disp	Z I C3 0	Springe, wenn: ST≠Quelloperand

7.3.2. REAL-Zahlen vergleichen und pop Stack

Befehl:	FCOMP	compare REAL and pop
Schreibweise:	FCOMP quelle FCOMP	

Exception-Flags: IE, DE
Condition-Code: C0, C3

Operandenkombinationen:

Typ	quelle
1	ST(i)
2	mem-op

Befehlswirkung:

Ein REAl-Operand im Stacktop wird mit einem REAL-Quelloperanden verglichen.
Der REAL-Quelloperand kann in einem beliebigen Stackregister, im Speicher als SHORT REAL- oder LONG REAL-Operand oder wenn kein Operand im Befehl angegeben wird, im Stacktop +1 ST(1) stehen.

Der Vergleich wird durchgefuehrt, indem der Quelloperand vom Inhalt des Stacktopregisters subtrahiert wird.
Das Vergleichsergebnis beeinflusst die CONDITION-Bits C3 und C0 im Statuswort.
Der Stackpointer wird nach der Operation inkrementiert.

```
Befehlsablauf: | ST - ST(i)                 |
               |_pop stack (ST<--ST+1)_| oder

               | ST - ST(1)                 |
               |_pop stack (ST<--ST+1)_| oder

               | ST - mem-op                |
               |_pop stack (ST<--ST+1)_|
```

Kodierung:

```
 ST-ST(i)_____________
|_11011000_|_11011(i)_|

 ST-REAL_________________________________________
|_11011m00_|_mod011r/m_|_disp.low_|_disp.high_|
 m=0 SHORT REAL, m=1 LONG REAL
```

Beispiele:

```
FCOMP  ST(3)
FCOMP  SOLLWERT
FCOMP
```

7.3.3. REAL-Zahlen vergleichen und pop Stack zweimal

Befehl:	FCOMPP	compare REAL and pop twice
Schreibweise:	FCOMPP	

Exception-Flags: IE, DE
Condition-Code: C0, C3

Befehlswirkung:

FCOMPP vergleicht den REAL-Operanden im Stacktop mit dem ersten Stackregister nach Stacktop ST(1), indem vom Stacktop ST(1) subtrahiert wird.

Das Vergleichsergebnis beeinflusst die CONDITION-Bits C3 und C0.

Nach der Operation wird der Stackpointer zweimal inkrementiert.

Befehlsablauf: ST - ST(1)
pop stack
pop stack (ST<--ST+2)

Kodierung:

11011110	11011001

Beispiel:

FCOMPP

7.3.4. INTEGER vergleichen

Befehl:	FICOM	INTEGER compare
Schreibweise:	FICOM quelle	

Exception-Flags: IE, DE
Condition-Code: C0, C3

Operandenkombinationen:

Typ	quelle
1	mem-op

Befehlswirkung:

Der Quelloperand steht als WORD- oder SHORT INTEGER-Zahl im Speicher.
Der Vergleichsbefehl konvertiert zunaechst den Speicheroperanden in das TEMPORARY REAL-Format.
Danach erfolgt der Vergleich mit dem Stacktopregister, indem der Speicheroperand vom Inhalt des Stacktopregisters subtrahiert wird.
Das Vergleichsergebnis beeinflusst die CONDITION-Bits C3 und C0.

Der Stackpointer bleibt unveraendert.

Befehlsablauf: ST - mem-op

Kodierung:

```
|_11011m10_|_mod010r/m_|_disp.low_|_disp.high_|
 m=0 SHORT, m=1 WORD INTEGER
```

Beispiel:

    FICOM ZAEHLER

7.3.5. INTEGER vergleichen und pop Stack

Befehl:	FICOMP	**INTEGER compare and pop**
Schreibweise:	FICOMP quelle	

Exception-Flags: IE, DE
Condition-Code: C0, C3

Operandenkombinationen:

Typ	quelle
1	mem-op

Befehlswirkung:

Der Quelloperand steht als WORD- oder SHORT INTEGER-Zahl im Speicher.
Der Vergleichsbefehl konvertiert zunaechst den Speicheroperanden in das TEMPORARY REAL-Format.
Danach erfolgt der Vergleich mit dem Stacktopregister, indem der Speicheroperand vom Inhalt des Stacktopregisters subtrahiert wird.
Das Vergleichsergebnis beeinflusst die CONDITION-Bits C3 und C0.

Der Stackpointer wird nach der Operation inkrementiert.

Befehlsablauf: ST - mem-op
pop stack (ST<--ST+1)

Kodierung:

```
|_11011m10_|_mod011r/m_|_disp.low_|_disp.high_|
 m=0 SHORT, m=1 WORD INTEGER
```

Beispiel:

    FICOMP ZUGANG

7.3.6. Stacktopelement testen

Befehl:	FTST	test stack top against +0.0
Schreibweise:	FTST	

Exception-Flags: IE, DE
Condition-Code: C0, C3

Befehlswirkung:

FTST vergleicht den Inhalt des Stacktopregisters mit +0.0. Dabei wird intern +0.0 vom Stacktopelement subtrahiert.

Das Vergleichsergebnis beeinflusst die CONDITION-Bits C3 und C0 im Statuswort.

Der Stackpointer bleibt unveraendert.

CONDITION-Code:

C3	C0	Interpretation
0	0	ST ist positiv
0	1	ST ist negativ
1	0	ST ist Null (+ oder -)
1	1	ST ist nicht vergleichbar (ST ist eine NAN oder projektiv oo)

Befehlsablauf: ST<--ST - 0.0

Kodierung:

| 11011001 | 11100100 |

Beispiel:

FTST

7.3.7. Stacktopelement pruefen

Befehl:	FXAM	examine stack top
Schreibweise:	FXAM	

Exception-Flags: keine Beeinflussung
Condition-Code: C0, C1, C2, C3

Befehlswirkung:

Der Befehl FXAM prueft den Inhalt des Stacktopregisters und gibt Auskunft ueber die Charakteristik des gespeicherten Operanden.

Der Befehl unterscheidet zwischen positiv/negativ, NAN/UNNORMAL/ DENORMAL/NORMAL/ZERO oder EMPTY.

Abhaengig vom Pruefergebnis wird in den CONDITION-Bits CØ - C3 ein bestimmter Kode abgelegt.

CONDITION-Code:

CONDITION-Code				Interpretation
C3	C2	C1	CØ	
Ø	Ø	Ø	Ø	+ UNNORMAL
Ø	Ø	Ø	1	+ NAN
Ø	Ø	1	Ø	- UNNORMAL
Ø	Ø	1	1	- NAN
Ø	1	Ø	Ø	+ NORMAL
Ø	1	Ø	1	+ oo
Ø	1	1	Ø	- NORMAL
Ø	1	1	1	- oo
1	Ø	Ø	Ø	+ Ø
1	Ø	Ø	1	EMPTY
1	Ø	1	Ø	- Ø
1	Ø	1	1	EMPTY
1	1	Ø	Ø	+ DENORMAL
1	1	Ø	1	EMPTY
1	1	1	Ø	- DENORMAL
1	1	1	1	EMPTY

Diese Tabelle zeigt die Interpetation aller CONDITION-Codes, die FXAM liefert.

Fuer EMPTY werden vier verschiedene Kodes geliefert, wobei die Bits C3 und CØ jeweils 1 sind.
Die Bits C2 und C1 koennen ignoriert werden, wenn das Stacktopregister auf EMPTY geprueft wird.

Kodierung:

| 11Ø11ØØ1 | 111ØØ1Ø1 |

Beispiel:

FXAM

7.4. Transzendentale Befehle

Uebersicht:

FPTAN	partial tangent
FPATAN	partial arctangent
F2XM1	2^X-1
FYL2X	$Y*\log_2 X$
FYL2XP1	$Y*\log_2(X+1)$

Mit diesen Befehlen koennen trigonometrische, invers trigonometrische, hyperbolische, invers hyperbolische, logarithmische und Exponential-Funktionen berechnet werden.

Eine transzendente Operation wird mit dem Stacktop durchgefuehrt. Das Operationsergebnis wird in den Stacktop geladen.

7.4.1. Partielle Tangensfunktion

Befehl:	FPTAN	**partial tangent**
Schreibweise:	FPTAN	

Exception-Flags: IE, PE

Befehlswirkung:

Die Funktion TAN (θ) = $\frac{Y}{X}$ wird berechnet.

Das Argument θ steht im Stacktop ST.

Die Funktion liefert als Ergebnis die REAL-Werte Y und X.

Dabei wird Y im Stacktop gespeichert, d.h. das Argument θ wird ueberschrieben.

Danach wird der Stackpointer dekrementiert, und X wird in den neuen Stacktop geladen.

Das Argument θ muss im Bereich von $0 < \theta < \frac{pi}{4}$ liegen.

Ist das Argument ungueltig (nicht normalisiert) oder liegt es ausserhalb des gueltigen Bereiches, liefert FPTAN ein undefiniertes Ergebnis, ohne eine Ausnahme zu signalisieren.

Befehlsablauf: Y/X <-- TAN(ST)
ST <-- Y
push stack (ST<--ST-1)
ST<-- X

Kodierung:

| 11011001 | 11110010 |

7.4.2. Partielle Arcustangensfunktion

Befehl:	FPATAN	**partial arctangent**
Schreibweise:	FPATAN	

Exception-Flags: UE, PE

Befehlsablauf:

Es wird die Funktion θ = arctan (X/Y) berechnet.

Im Stacktop ST ist X und im Stacktop + 1 ST(1) ist Y gespeichert.

X und Y muessen die Bedingung 0 < Y < X < +oo erfuellen.

Bei der Berechnung der Funktion wird das Ergebnis in das TEMPORARY REAL-Format konvertiert, danach der Stackpointer inkrementiert und die Funktion in den neuen Stacktop geladen.

Ist das Argument ungueltig (nicht normalisiert) oder liegt es ausserhalb des gueltigen Bereiches, liefert FPATAN ein undefiniertes Ergebnis, es wird keine Ausnahme signalisiert.

Befehlsablauf: TEMPORARY REAL T1<--arctan (ST(1)/ST)
pop stack (ST<--ST+1)
ST<--T1

Kodierung:

| 11011001 | 11110011 |

7.4.3. $2^X - 1$ berechnen

Befehl:	F2XM1	2^X-1
Schreibweise:	F2XM1	

Exception-Flags: UE, PE

Befehlswirkung:

Bei der Berechnung der Funktion $Y = 2^X-1$ wird X dem Stacktopregister entnommen.

Das Ergebnis Y ueberschreibt X, der Stacktop bleibt unveraendert.

X muss im Bereich 0 <= X <= 0.5 liegen.

Der Befehl liefert ein sehr genaues Ergebnis, wenn X nahe 0 liegt.

Um $Y = 2^X$ zu erhalten, ist zum Ergebnis, das F2XM1 liefert, +1.0 zu addieren.

Befehlsablauf: ST <-- 2^{ST} -1

Kodierung:

11011001	11110000

Beachte:

Folgende Formeln koennen verwendet werden, um die Exponentialfunktionen zu anderen Basen als 2 zu berechnen:

$$10^X = 2^{X*Log_2 10}$$

$$e^X = 2^{X*Log_2 e}$$

$$Y^X = 2^{X*Log_2 Y}$$

Der WM87 hat Befehle, die die Konstanten $Log_2 10$ und $Log_2 e$ laden.

Weiterhin bietet der Befehl FYL2X die Moeglichkeit, die Funktion $x*Log_2 Y$ zu berechnen.

7.4.4. Y * Log_2X berechnen

Befehl:	FYL2X	Y * Log_2X
Schreibweise:	FYL2X	

Exception-Flags: PE

Befehlswirkung:

Der Befehl berechnet die Funktion Z = Y * Log_2X.

X wird dem Stacktopregister und Y dem Stacktopregister +1 ST(1) entnommen.
Das Produkt ST(1) * Log_2ST wird gebildet und in das TEMPORARY REAL-Format umgewandelt.
Dann wird der Stackpointer inkrementiert und das TEMPORARY REAL-Format in den neuen Stacktop geladen.

Die Operanden X und Y muessen im Bereich liegen:
Ø < X < oo und -oo < Y < +oo

Befehlsablauf: TEMPORARY REAL T1<--ST(1) * Log_2(ST)
pop stack
ST<--T1

Kodierung:

| 11011001 | 11110001 |

7.4.5. Y * Log_2 (X+1) berechnen

Befehl:	FYL2XP1	Y * Log_2 (X+1)
Schreibweise:	FYL2XP1	

Exception-Flags: PE

Dieser Befehl berechnet die Funktion Z = Y * Log_2 (X+1), X ist im Stacktopregister und Y im Stacktopregister +1 ST^2(1) gespeichert.

X muss im Bereich Ø < |X| < (1 -\/¯2¯ /2) liegen,
Y muss im Bereich -oo < Y < oo liegen.

Zum Inhalt X wird 1 addiert und die Summe in das TEMPORARY REAL-Format T1 konvertiert.

Danach wird ST(1) = Y mit Log_2 T1 multipliziert und das Produkt in das TEMPRARY REAL-Format T2 konvertiert.

Der Befehl FYL2XP1 inkrementiert anschliessend den Stackpointer und schreibt die Funktion Z(T2) = Y * Log_2 (X+1) in den neuen Stacktop.

Dieser Befehl liefert fuer die Berechnung des Logarithmus von Zahlen, die sehr nahe bei 1 liegen, eine verbesserte Genauigkeit als der Befehl FYL2X.

Befehlsablauf:

```
TEMPORARY REAL T1<--ST +1
TEMPORARY REAL T2<--ST(1) * Log2 T1
pop stack (ST<--ST+1)
ST<--T2
```

Kodierung:

```
 ---------------------
| 11011001 | 11111001 |
 ---------------------
```

Beispiel:

Es soll Z = Y * Log_2 (1.0000378) berechnet werden.

In diesem Fall gilt: 1.0000378 = 1 + 0.0000378 = 1 + E mit E << 1.

X = E = 0.0000378 ist im Stacktop zu speichern, um mit FYL2XP1 die Funktion Z = Y * Log_2 (0.0000378 + 1) = Y * Log_2 (1.0000378) zu berechnen.

7.5. Konstanten

Uebersicht:

FLDZ	load +0.0
FLD1	load +1.0
FLDPI	load Pi
FLDL2T	load $log_2 10$
FLDL2E	load $log_2 e$
FLDLG2	load $log_{10} 2$
FLDLN2	load $log_e 2$

Diese Befehle laden (push) die gewaehlte Konstante in den Stacktop.
Alle Konstanten haben die TEMPORARY REAL-Praezision von 64 Bits.

7.5.1. +0.0 in den Stacktop laden

Befehl:	FLDZ	load +0.0
Schreibweise:	FLDZ	

Exception-Flags: IE

Befehlswirkung:

Zuerst wird der Stackpointer dekrementiert und danach die Konstante 0.0 in das neue Stacktopregister geladen.

Befehlsablauf: push stack (ST<--ST-1)
ST<-- +0.0

Kodierung:

| 11011001 | 11101110 |

7.5.2. +1.0 in den Stacktop laden

Befehl:	FLD1	load +1.0
Schreibweise:	FLD1	

Exception-Flags: IE

Befehlswirkung:

Der Stackpointer wird dekrementiert und anschliessend die Konstante 1.0 in das neue Stacktopregister geladen.

Befehlsablauf: push stack (ST<--ST-1)
ST<-- +1.0

Kodierung:

| 11011001 | 11101000 |

7.5.3. Pi in den Stacktop laden

Befehl:	FLDPI	load pi
Schreibweise:	FLDPI	

Exception-Flags: IE

Befehlswirkung:

Die Konstante Pi hat die TEMPORARY-REAL-Praezision von 64 Bits und eine Genauigkeit von 19 Dezimalstellen.

Der Stackpointer wird dekrementiert und dann die Konstante Pi in das neue Stacktopregister geladen.

Befehlsablauf: push stack (ST<--ST-1)
ST<-- Pi

Kodierung:

11011001	11101011

7.5.4. $Log_2 10$ in den Stacktop laden

Befehl:	FLDL2T	load $Log_2 10$
Schreibweise:	FLDL2T	

Exception-Flags: IE

Befehlswirkung:

Die Konstante $Log_2 10$ hat die TEMPORARY REAL-Praezision von 64 Bits und eine Genaufgkeit von 19 Dezimalstellen.

Der Stackpointer wird dekrementiert und danach die Konstante $Log_2 10$ in das neue Stacktopregister geladen.

Befehlsablauf: push stack (ST<--ST-1)
ST<-- $Log_2 10$

Kodierung:

11011001	11101001

7.5.5. Log_2e in den Stacktop laden

Befehl:	FLDL2E	load Log_2e
Schreibweise:	FLDL2E	

Exception-Flags: IE

Befehlswirkung:

Der Stackpointer wird dekrementiert und dann die Konstante Log_2e in das neue Stacktopregister geladen.

Befehlsablauf: push stack (ST<--ST-1)
ST<--Log_2e

Kodierung:

| 11011001 | 11101010 |

7.5.6. $Log_{10}2$ in den Stacktop laden

Befehl:	FLDLG2	load $Log_{10}2$
Schreibweise:	FLDLG2	

Exception-Flags: IE

Befehlswirkung:

Die Konstante hat die TEMPORARY REAL-Praezision von 64 Bits und eine Genauigkeit von 19 Dezimalstellen.

Der Stackpointer wird dekrementiert und die Konstante $Log_{10}2$ in das neue Stacktopregister geladen.

Befehlsablauf: push stack (ST<--ST-1)
ST<--$Log_{10}2$

Kodierung:

| 11011001 | 11101100 |

7.5.7. Log_e2 in den Stacktop laden

Befehl:	FLDLN2	load Log_e2
Schreibweise:	FLDLN2	

Exception-Flags: IE

Befehlswirkung:

Die Konstante hat die TEMPORARY REAL-Praezision von 64 Bits und eine Genauigkeit von 19 Dezimalstellen.

Der Stackpointer wird zuerst dekrementiert und dann die Konstante Log_e2 in das neue Stacktopregister geladen

Befehlsablauf: push stack (ST<--ST-1)
ST<--log_e2

Kodierung:

| 11011001 | 11101101 |

7.6. Prozessorsteuerbefehle

Uebersicht:

FINIT/FNINIT	initialize processor
FDISI/FNDISI	disable interrupts
FENI/FNENI	enable interrupts
FLDCW	load control word
FSTCW/FNSTCW	store control word
FSTSW/FNSTSW	store status word
FCLEX/FNCLEX	clear exceptions
FSTENV/FNSTENV	store environment
FLDENV	load environment
FSAVE/FNSAVE	save state
FRSTOR	restore state
FINCSTP	increment stack pointer
FDECSTP	decrement stack pointer
FFREE	free register
FNOP	no operation
FWAIT	CPU wait

Diese Befehle werden zum Laden und Speichern von Zustaenden aus der Hard- und Softwareumgebung des WM87 verwendet.

Der WM87 kann durch diese Befehle in einen definierten Anfangszustand gebracht werden (Initialisieren), es koennen Ausnahmezustaende (Exceptions) behandelt werden, und der WM87 kann fuer verschiedene Tasks vorbereitet werden (task switching).

Bei einigen Prozessorsteuerbefehlen kann eine WAIT- oder NOWAIT-Form kodiert werden.
Die NOWAIT-Form wird durch das Zeichen "N" in der Mnemonik des Befehles angegeben. Dann generiert der Assembler das CPU-WAIT-Prefix nicht.
Die NOWAIT-Formen werden verwendet, um in kritischen Koderegionen einen endlosen Wartezustand zu vermeiden.

Die WAIT-Formen der Prozessorsteuerbefehle sollten in Koderegionen verwendet werden, in denen die Interruptanforderungen zur CPU freigegeben sind.

Die Prozessorsteuerbefehle werden in der "Control Unit" (CU) des WM87 bearbeitet. Sie koennen ausgefuehrt werden, waehrend in der "Numeric Execution Unit"(NEU) ein Nichtsteuerbefehl ablaeuft.
Um zu gewaehrleisten, dass ein Prozessorsteuerbefehl erst ausgefuehrt wird, wenn die Operation in der NEU abgeschlossen ist, sollte die WAIT-Form des Steuerbefehles verwendet werden.

Eine Ausnahme sind die "Store Enviroment-" und "Save Status"-Befehle. Die NOWAIT-Formen FNSTENV und FNSAVE werden erst nach Abschluss der Operation in der NEU ausgefuehrt.

7.6.1. Prozessor Initialisieren

Befehl:	FINIT / FNINIT	**initialize processor**
Schreibweise:	FINIT / FNINIT	

Exception-Flags: keine Beeinflussung

Befehlswirkung:

Der WM87 wird initialisiert, diese Befehle sind einem Hardware-Reset aequivalent.

Die Softwareinitialisierung beeinflusst im Gegensatz zur Hardwareinitialisierung das "Tracking" (Gleichlauf) der CPU durch den WM87 nicht.

Kodierung:

11011011	11100011

Prozessorstatus nach dem Initialisieren:

Feld	Wert	Interpretation
Kontrollwort:		
INFINITY CONTROL	0	Projektives Modell
ROUNDING CONTROL	00	Runden zum naechsten Wert
PRECISION CONTROL	11	64 Bits
INTERRUPT-ENABLE MASK	1	Interrupt sperren
EXCEPTION MASKS	111111	Alle Exceptions maskiert
Statuswort:		
BUSY	0	Nicht BUSY
CONDITION CODE	????	Unbestimmt
STACKTOP POINTER	000	Stackpointer geloescht
INTERRUPT REQUEST	0	Kein Interrupt
EXCEPTION FLAGS	000000	Keine Exceptions
Identifizierungswort:		
TAG(7...0)	11	Die Stackregister werden als EMPTY gekennzeichnet.
Register:	N.C.	Unbeeinflusst (Not Changed)
Exception Pointers:		
Befehlskode	N.C.	Unbeeinflusst
Befehlsadresse	N.C.	Unbeeinflusst
Operandenadresse	N.C.	Unbeeinflusst

Der Assembler generiert bei FINIT bei der Uebersetzung den Kode des WM86-WAIT-Befehles.
Beim FNINIT wird dieser Kode nicht erzeugt.

7.6.2. Interruptsperre

Befehl:	FDISI / FNDISI	disable interrupts
Schreibweise:	FDISI / FNDISI	

Exception-Flags: keine Beeinflussung

Befehlswirkung:

FDISI setzt im Kontrollwort das INTERRUPT ENABLE-Maskenbit (IEM=1) nach WAIT, beim Befehl FNDISI ohne WAIT.

Eine Interruptanforderung durch den WM87 wird dadurch verhindert.

Beim Befehl FDISI generiert der Assembler bei der Uebersetzung zusaetzlich den Kode des WM86-WAIT-Befehles, beim FNDISI wird der Kode nicht erzeugt.

Der FNDISI wird in kritischen Kodebereichen verwendet, bei denen ein WAIT-Befehl eine Systemblockade verursachen kann.

Kodierung:

| 11011011 | 11100001 |

Beachte:

Wird bei einer anstehenden Exception die WAIT-Form des Befehles dekodiert, generiert der WM87 eine Interruptanforderung (INT=HIGH) unabhaengig davon, ob die INTERRUPT-ENABLE-Maske gesetzt ist oder nicht.

7.6.3. Interruptfreigabe

Befehl:	FENI / FNENI	enable interrupts
Schreibweise:	FENI / FNENI	

Exception-Flags: keine Beeinflussung

Befehlswirkung:

FENI loescht im Kontrollwort das INTERRUPT ENABLE-Maskenbit (IEM=0) nach WAIT, beim Befehl FNENI ohne WAIT.

Dem WM87 wird dadurch erlaubt, eine Interruptanforderung zu generieren (unmaskierte Exception).

Bei FENI wird bei der Uebersetzung durch den Assembler zusaetzlich der Kode des WAIT-Befehles generiert, bei FNENI wird der Kode nicht erzeugt.

FENI wird in kritischen Kodebereichen verwendet, bei denen ein WAIT-Befehl eine Systemblockade verursachen kann.

FENI wird verwendet, um sicher zu sein, dass er erst nach der Bearbeitung des vorherigen Befehles in der NEU ausgefuehrt wird.

Kodierung:

| 11011011 | 11110000 |

7.6.4. Kontrollwort laden

Befehl:	FLDCW	load control word
Schreibweise:	FLDCW quelle	

Exception-Flags: keine Beeinflussung

Operandenkombinationen:

Typ	quelle
1	mem-op

Befehlswirkung:

Mit dem Befehl FLDCW ist die Betriebsart des WM87 programmierbar, das aktuelle Prozessorkontrollwort wird von dem Quelloperanden ueberschrieben.

Der Quelloperand steht im Speicher und ist vom Typ WORD INTEGER.

Der Stackpointer bleibt unveraendert.

Befehlsablauf: WM87-Kontrollwort <-- mem-op

Kodierung:

11011001	mod101r/m	disp.low	disp.high

Beachte:

Ist ein EXCEPTION-Flag im Statuswort gesetzt, kann mit diesem Befehl ein neues Kontrollwort geladen werden, um die Exceptions zu demaskieren.
Dadurch kommt es vor dem Ausfuehren des folgenden Befehles zur Interruptausloesung (Pollingbetrieb).

Soll die Betriebsart des WM87 geaendert werden, wird empfohlen, die Exceptions zuerst zu loeschen (FCLEX) und dann das Kontrollwort zu laden.

Beispiel:

```
FLDCW   KONTROLLWORT
```

7.6.5. Kontrollwort speichern

Befehl:	FSTCW / FNSTCW	**store control word**
Schreibweise:	FSTCW ziel FNSTCW ziel	

Exception-Flags: keine Beeinflussung

Operandenkombinationen:

Typ	ziel
1	mem-op

Befehlswirkung:

Das aktuelle Prozessorkontrollwort wird in den Speicher geschrieben, beim Befehl FSTCW nach WAIT, bei Befehl FNSTCW ohne WAIT.
Der Assembler generiert bei FSTCW bei der Uebersetzung zusaetzlich den Kode des WM86-WAIT-Befehles, beim FNSTCW wird dieser Kode nicht generiert.

Der Befehl FNSTCW wird in kritischen Kodebereichen verwendet, in denen ein WAIT-Befehl eine Systemblockade verursachen kann.

Befehlsablauf: mem-op <-- Kontrollwort

Kodierung:

11011001	mod111r/m	disp.low	disp.high

Beispiel:

```
FSTCW  KONTR_SPEI
```

7.6.6. WM87-Statuswort speichern

Befehl:	FSTSW / FNSTSW	**store status word**
Schreibweise:	FSTSW ziel FNSTSW ziel	

Exception-Flags: keine Beeinflussung

Operandenkombinationen:

Typ	ziel
1	mem-op

Befehlswirkung:

Die im Statuswort gespeicherten Informationen werden in den Speicher geschrieben, beim Befehl FSTSW nach WAIT und bei FNSTSW ohne WAIT.
Der Zielspeicheroperand ist vom Typ WORD INTEGER .

Die Statusinformationen Bit 0 bis Bit 7 werden unter der niederwertigen Adresse und die Statusinformationen Bit 8 bis Bit 15 unter der hoeherwertigen Adresse im Speicher abgelegt.

Der Stackpointer bleibt unbeeinflusst.

Befehlsablauf: mem-op <-- WM87-Statuswort

Kodierung:

11011101	mod111r/m	disp.low	disp.high

Beispiel:

FSTSW STATUS87

7.6.7. Exceptions loeschen

Befehl:	FCLEX / FNCLEX	**clear exceptions**
Schreibweise:	FCLEX / FNCLEX	

Exception-Flags: IE, DE, ZE, OE, UE, PE

Befehlswirkung:

Diese Befehle loeschen im Statuswort alle EXCEPTION-Flags, das INTERRUPT REQUEST-Flag (IR) und das BUSY-Flag, der Befehl FCLEX nach WAIT und FNCLEX ohne WAIT.
Die WM87 INT- und BUSY-Leitungen gehen dabei in den inaktiven Zusatand (LOW).
Beim Befehl FCLEX generiert der Assembler bei der Uebersetzung zusaetzlich den Kode des WM86-WAIT-Befehles, bei FNCLEX wird der Kode nicht erzeugt.

FNCLEX wird in kritischen Kodebereichen verwendet, bei denen ein WAIT-Befehl eine Systemblockade verursachen kann.

Der FCLEX-Befehl wird verwendet, um sicherzugehen, dass dieser erst nach der Bearbeitung eines vorherigen Befehles in der NEU ausgefuehrt wird.

Der Stackpointer bleibt unbeeinflusst.

Kodierung:

11011011	11100010

7.6.8. Umgebung speichern

Befehl:	FSTENV / FNSTENV	store environment
Schreibweise:	FSTENV ziel FNSTENV ziel	

Exception-Flags: keine Beeinflussung

Operandenkombinationen:

Typ	ziel
1	mem-op

Befehlswirkung:

Diese Befehle schreiben den WM87-Basisstatus (Kontrollwort, Statuswort, Tag Word) und die Exception Pointers (Befehlsadresse, Befehlskode, Operandenadresse) in den Speicher.

Ueblicherweise werden diese Informationen in den Stack der CPU geschrieben.

Alle Exception-Masken im Kontrollwort des WM87 werden nach der Operation gesetzt (Mask =1).

Eine Ausnahme bildet die INTERRUPT ENABLE-Maske (IEM), sie bleibt unbeeinflusst.

Beim Befehl FSTENV generiert der Assembler bei der Uebersetzung den Kode des Wait-Befehles, beim FNSTENV wird dieser Kode nicht erzeugt.

Befehlsablauf: mem-op <-- WM87-Umgebung

Kodierung:

11011001	mod110r/m	disp.low	disp.high

Beispiel:

FSTENV [BP]

Beachte:

Wird waehrend der Ausfuehrung eines WM87-Befehles in der NEU der Befehl FNSTENV in der CU dekodiert, speichert der WM87 seine Umgebung solange nicht, bis der Befehl in der NEU vollstaendig ausgefuehrt ist. Dadurch wird gewaehrleistet, dass die aktuelle Umgebung des WM87 gespeichert wird.
Waehrend FSTENV/FNSTENV die WM87-Umgebung in den Speicher schreiben, duerfen andere WM87-Befehle nicht dekodiert werden.
Beim Befehl FSTENV sollte ein vom Assembler generierter WAIT dem nachfolgenden WM87-Befehl vorausgehen.
Hat der nachfolgende WM87-Befehl die NOWAIT-Form, ist vor diesem Befehl FWAIT zu schreiben.
Wird FNSTENV bei gesperrten CPU-Interrupts ausgefuehrt, sollte, bevor die Sperre fuer die CPU-Interrupts aufgehoben wird, ein FWAIT-Befehl ausgefuehrt werden.
Es besteht keine Gefahr, dass FWAIT zu einem endlosen Wartezustand fuehrt, da FSTENV und FNSTENV alle Exceptions maskiert, so dass eine Interruptanforderung durch den WM87 verhindert wird.

7.6.9. Umgebung laden

Befehl:	FLDENV	load environment
Schreibweise:	FLDENV quelle	

Exception-Flags: keine Beeinflussung

Operandenkombinationen:

Typ	quelle
1	mem-op

Befehlswirkung:

Mit dem Befehl FLDENV kann die WM87-Umgebung wieder in den WM87 zurueckgeladen werden, nachdem sie mit den Befehlen FSTENV/FNSTENV in den Speicher geschrieben wurde.
Diesem Befehl duerfen CPU-Befehle unmittelbar folgen.

Ein nachfolgender WM87-Befehl sollte mit einem FWAIT oder mit einem vom Assembler generierten WAIT ausgefuehrt werden.

Befehlsablauf: WM87-Umgebung <-- mem-op

Kodierung:

11011001	mod100r/m	disp.low	disp.high

Beispiel:

FLDENV [BP+6]

Beachte:

Wird eine Umgebung geladen, die eine unmaskierte Exception enthaelt, wird eine unmittelbare Interruptanforderung durch den WM87 generiert (vorausgesetzt IEM = 0).

7.6.10. Status speichern

Befehl:	FSAVE / FNSAVE	**save state**
Schreibweise:	FSAVE ziel FNSAVE ziel	

Exception-Flags: keine Beeinflussung

Operandenkombinationen:

Typ	ziel
1	mem-op

Befehlswirkung:

Der komplette WM87-Status (Prozessorumgebung einschliesslich aller Stackregister) wird in den Speicher geschrieben.

Nach der Operation wird der WM87 neu initialisiert.

Beim FSAVE-Befehl generiert der Assembler bei der Uebersetzung den Kode des WAIT-Befehles, bei FNSAVE wird dieser Kode nicht generiert.
Normalerweise wird der WM87-Status in den Stack der CPU geschrieben.

Beide Befehle werden verwendet, wenn in einem Programm der augenblickliche Status des WM87 gerettet werden muss, um anschliessend

den Prozessor fuer eine neue Routine zu initialisieren.

Befehlsablauf: mem-op <-- WM87-Status

Kodierung:

```
 ---------------------------------------------
|_11011101_|_mod110r/m_|_disp.low_|_disp.high_|
```

Beispiel:

FSAVE [BP]

Speicherinhalt nach FSAVE/FNSAVE:

```
                     15                                       0
                     ------------------------------------------
                                                              |
                       CONTROL WORD                           |   + 0
                       STATUS WORD                            |   + 2
                       TAG WORD                               |   + 4
INSTRUCTION      |‾  IP15 - 0                                 |   + 6
POINTER          |_  IP19 - 16       | 0 |     OPCODE         |   + 8
                                                              |
DATA (OPERAND)   |‾  OP15 - 0                                 |   +10
POINTER          |_  OP19 - 16       |            0           |   +12
                                                              |
                 |‾   SIGNIFIKANT 15 - 0                      |   +14
                 |    SIGNIFIKANT 31 - 16                     |   +16
TOP STACK        |    SIGNIFIKANT 47 - 32                     |   +18
ELEMENT: ST      |    SIGNIFIKANT 63 - 48                     |   +20
                 |  ^                                         |
                 |_S| EXPONENT 14 - 0                         |   +22
                                                              |
                 |‾   SIGNIFIKANT 15 - 0                      |   +24
                 |    SIGNIFIKANT 31 - 16                     |   +26
NAECHSTES STACK  |    SIGNIFIKANT 47 - 32                     |   +28
ELEMENT: ST(1)   |    SIGNIFIKANT 63 - 48                     |   +30
                 |  ^                                         |
                 |_S| EXPONENT 14 - 0                         |   +32
                      .                                       |
                      .
                      .                                       |
                 |‾   SIGNIFIKANT 15 - 0                      |   +84
                 |    SIGNIFIKANT 31 - 16                     |   +86
LETZTES STACK-   |    SIGNIFIKANT 47 - 32                     |   +88
ELEMENT: ST(7)   |    SIGNIFIKANT 63 - 48                     |   +90
                 |  ^                                         |
                 |_S| EXPONENT 14 - 0                         |   +92
```

S = Vorzeichen

FSAVE und FNSAVE schreiben 94 Bytes in den Speicher.

Beachte:

Wird waehrend der Ausfuehrung eines WM87-Befehles in der NEU der Befehl FNSAVE in der CU dekodiert, speichert der WM87 seinen Status so lange nicht, bis der Befehl in der NEU vollstaendig ausgefuehrt ist.
Dadurch wird gewaehrleistet, dass der aktuelle Status des WM87 gespeichert wird.
Waehrend FSAVE/FNSAVE den WM87-Status in den Speicher schreiben, duerfen andere WM87-Befehle nicht dekodiert werden.
Es sollte deshalb ein vom Assembler generierter WAIT dem nachfolgenden WM87-Befehl vorausgehen.
Hat der nachfolgende WM87-Befehl die NOWAIT-Form, sollte vor diesem ein FWAIT-Befehl stehen.
Wird FNSAVE bei gesperrten CPU-Interrupts ausgefuehrt, sollte, bevor die Sperre fuer die CPU-Interrupts aufgehoben wird, ein FWAIT ausgefuehrt werden.
Da FNSAVE den WM87 initialisiert, besteht keine Gefahr, dass FWAIT zu einem endlosen Wartezustand fuehrt.

7.6.11. Status laden

Befehl:	**FRSTOR**	**restore saved state**
Schreibweise:	**FRSTOR quelle**	

Exception-Flags: keine Beeinflussung

Operandenkombinationen:

Typ	quelle
1	mem-op

Befehlswirkung:

Der Befehl laedt den Status des WM87 aus dem 94-Byte-Speicherbereich zurueck.
CPU-Befehle koennen diesem Befehl unmittelbar folgen.

Ein nachfolgender WM87-Befehl sollte mit einem FWAIT oder einem vom Assembler generierten WAIT ausgefuehrt werden.

Beachte:

Nach Beendigung von FRSTOR wird der WM87 unmittelbar eine Interruptanforderung generieren, falls der Status eine Exception enthaelt und IEM = 0 ist.

Befehlsablauf: WM87-Status <-- mem-op

Kodierung:

| 11011101 | mod100r/m | disp.low | disp.high |

Beispiel:

FRSTOR [BP-94]

7.6.12. Stackpointer inkrementieren

Befehl:	FINCSTP	increment stack pointer
Schreibweise:	FINCSTP	

Exception-Flags: keine Beeinflussung

Befehlswirkung:

Der FINCSTP-Befehl addiert zum Inhalt des ST-Feldes im Statuswort die Zahl "1".
Dabei werden weder TAG-Felder noch Registerinhalte geaendert.

Dieser Befehl ist einem POP-Befehl nicht aequivalent, da er das TAG-Feld des vorherigen Stackregisters nicht mit dem Kode fuer EMPTY besetzt.

Kodierung:

| 11011001 | 11110111 |

7.6.13. Stackpointer dekrementieren

Befehl:	FDECSTP	decrement stack pointer
Schreibweise:	FDECSTP	

Exception-Flags: keine Beeinflussung

Befehlswirkung:

FDECSTP subtrahiert vom Inhalt des ST-Feldes im Statuswort eine "1". Dabei werden weder TAG-Felder noch Registerinhalte geaendert.

Kodierung:

| 11011001 | 11110110 |

7.6.14. Stackelement als EMPTY markieren

Befehl:	FFREE	**free register**
Schreibweise:	FFREE ziel	

Exception-Flags: keine Beeinflussung

Operandenkombinationen:

Typ	ziel
1	ST(i)

Befehlswirkung:

Der Befehl markiert das i-te Stackregister nach dem Topstack = Zielelement ST(i) als EMPTY.
Das entsprechende TAG-Feld wird deshalb mit der Bitkombination
TAG(i) besetzt
11

Der Inhalt des Zielelementes bleibt unbeeinflusst.

Der Stackpointer bleibt unbeeinflusst.

Kodierung:

| 11011101 | 11000(i) |

Beispiel:

FFREE ST(3)

7.6.15. Keine Operation

Befehl:	FNOP	**no operation**
Schreibweise:	FNOP	

Exception-Flags: keine Beeeinflussung

Befehlswirkung:

Der Befehl speichert das Stacktopregister ST im Stacktopregister, d.h. es wird keine effektive Operation ausgefuehrt.

FNOP ist zur Bildung von Verzoegerungszeiten sinnvoll.

Der Stackpointer bleibt unbeeinflusst.

Befehlsablauf: ST<--ST

Kodierung:

| 11011001 | 11010000 |

7.6.16. CPU WAIT

Befehl:	FWAIT	(CPU) wait while WM87 is busy
Schreibweise:	FWAIT	

Exception-Flags: keine Beeinflussung

Befehlswirkung:

Dieser Befehl ist eine alternative Mnemonik fuer den CPU-WAIT-Befehl.

FWAIT sollte verwendet werden, wenn die CPU mit dem WM87 synchronisiert werden soll, d.h. dass weitere Befehle solange nicht dekodiert werden, bis der WM87 die Ausfuehrung seines augenblicklichen Befehles beendet hat.

Kodierung:

| 10011011 |

Beispiel:

Die CPU moechte einen Operanden untersuchen, der vom WM87 z.B. mit dem FIST-Befehl gespeichert wurde.
In diesem Fall wird ein nachfolgender FWAIT-Befehl die CPU solange in Wartezustand bringen, bis der WM87 den Operanden in den Speicher geschrieben hat.

8. Befehlsuebersicht

Datentransferbefehle				
Laden Speicher zum ST				
(FLD/FILD/FBLD)				
REAL/INTEGER	escmf1	mod000r/m	disp.low	disp.high
LONG INTEGER	esc111	mod101r/m	disp.low	disp.high
TEMPORARY REAL	esc011	mod101r/m	disp.low	disp.high
BCD	esc111	mod100r/m	disp.low	disp.high
ST(1) zu ST	esc001	11000ST(1)		
Laden ST zum Speicher				
(FST/FIST)				
ST zu REAL/INTEGER	escmf1	mod010r/m	disp.low	disp.high
ST zu ST(i)	esc101	11010ST(i)		
Laden ST zum Speicher und pop				
(FSTP/FISTP/FBSTP				
ST zu REAL/INTEGER	escmf1	mod011r/m	disp.low	disp.high
ST zu LONG INTEGER	esc111	mod111r/m	disp.low	disp.high
ST zu TEMPORARY REAL	esc011	mod111r/m	disp.low	disp.high
ST zu BCD	esc111	mod110r/m	disp.low	disp.high
ST zu ST(i)	esc101	11011ST(i)		
Vertauschen ST, ST(i)				
(FXCH)				
ST und ST(i)	esc001	11001ST(i)		
Arithmetische Befehle				
Addition Speicher mit ST				
(FADD/FIADD/FADDP)				
REAL/INTEGER mit ST	escmf0	mod000r/m	disp.low	disp.high
ST(i) mit ST	escdp0	11000ST(i)		
Subtraktion Speicher mit ST				
(FSUB/FISUB/FSUBP/FSUBR/FISUBR/FSUBRP)				
REAL/INTEGER und ST	escmf0	mod10rr/m	disp.low	disp.high
ST(i) und ST	escdp0	1110rr/m		
Multiplikation Speicher mit ST				
(FMUL/FIMUL/FMULP)				
REAL/INTEGER mit ST	escmf0	mod001r/m	disp.low	disp.high
ST(i) mit ST	escdp0	11001r/m		
Division Speicher mit ST				
(FDIV/FIDIV/FDIVP/FDIVR/FIDIVR/FDIVRP)				
REAL/INTEGER mit ST	escmf0	mod11rr/m	disp.low	disp.high
ST(i) mit ST	escdp0	1111rr/m		

Weitere Operationen

Wurzel von ST (FSQRT)	esc001	11111010
$z = y * 2^x$ (FSCALE)	esc001	11111101
Modulo-Div. (FPREM)	esc001	11111000
zu INT.runden (FRNDINT)	esc001	11111100
Trennen Exp.u.Signif. (FXTRACT)	esc001	11110100
Absolutwert ST (FABS)	esc001	11100001
Vorz. wechseln (FCHS)	esc001	11100000

Vergleichsbefehle

Vergleichen ST mit Speicher
(FCOM/FICOM)

REAL/INTEGER mit ST	escmf0	mod010r/m	disp.low	disp.high
ST(i) mit ST	esc000	11010ST(i)		

Vergleichen ST mit Speicher und pop
(FCOMP/FICOMP)

REAL/INTEGER mit ST	escmf0	mod011r/m	disp.low	disp.high
ST(i) mit ST	esc000	11011ST(i)		

Weitere Operationen

Vergleichen ST(1) mit ST u. pop zweimal (FCOMPP)	esc110	11011001
Testen ST (FTST)	esc001	11100100
Pruefen ST (FXAM)	esc001	11100101

Transzendentale Befehle

Part. Tangens (FPTAN)	esc001	11110010
Part. Arcustang. (FPATAN)	esc001	11110011
$y = 2^x-1$ (F2XM1)	esc001	11110000
$z = y*Log_2 x$ (FYL2X)	esc001	11110001
$z = y*Log_2(x+1)$ (FYL2XP1)	esc001	11111001

Konstanten

Laden nach ST

+0.0	(FLDZ)	esc001	11101110
+1.0	(FLD1)	esc001	11101000
Pi	(FLDPI)	esc001	11101011
$Log_2 10$	(FLDL2T)	esc001	11101010
$Log_2 e$	(FLDL2E)	esc001	11101010
$Log_{10} 2$	(FLDLG2)	esc001	11101100
$Log_e 2$	(FLDLN2)	esc001	11101101

Prozessorsteuerbefehle

Initialisieren				
(FINIT/FNINIT)	esc011	11100011		
Interrupt:				
Sperre (FDISI/FNDISI)	esc011	11100000		
Freigabe (FENI/FNENI)	esc011	11100001		
Kontrollwort:				
Laden (FLDCW)	esc001	mod101r/m	disp.low	disp.high
Speichern (FSTCW/FNSTCW)	esc001	mod111r/m	disp.low	disp.high
Statuswort:				
Speichern (FSTSW/FNSTSW)	esc101	mod111r/m	disp.low	disp.high
Exception:				
Loeschen (FCLEX/FNCLEX)	esc011	11100010		
Umgebung:				
Speichern(FSTENV/FNSTENV)	esc001	mod110r/m	disp.low	disp.high
Laden (FLDENV)	esc001	mod100r/m	disp.low	disp.high
Status:				
Speichern (FSAVE/FNSAVE)	esc101	mod110r/m	disp.low	disp.high
Laden (FRSTOR)	esc101	mod100r/m	disp.low	disp.high
Stackpointer:				
Inkrementieren (FINCSTP)	esc001	11110111		
Dekrementieren (FDECSTP)	esc001	11110110		
ST(i) leer (FFREE)	esc101	11000ST(i)		
Keine Operation (FNOP)	esc001	11010000		
CPU WAIT auf WM87(FWAIT)	10011011			

Erklaerungen:

mf = Speicherformat (memory format)
00 = 32-Bit REAL
01 = 32-Bit INTEGER
10 = 64-Bit REAL
11 = 16-Bit INTEGER

ST = augenblickliche Spitze des Gleitpunktstack (Stacktop)
ST(i) = i-tes Stackelement nach Stacktop

d = Ziel (destination)
0 = ST
1 = ST(1)

r = umgekehrt (reverse)
0 = Ziel (Operation) Quelle
1 = Quelle (Operation) Ziel

Ausnahmen_und_ihre_Antworten

Bedingung	maskierte Antwort	unmaskierte Antwort
u n g u e l t i g e	O p e r a t i o n	
Quellregister ist als EMPTY (Tag W.) gekennzeichnet.	Lese REAL INDEFINITE	Interruptanforderung
Zielregister ist nicht als EMPTY (Tag Word) gekennzeichnet.	Zielregister ueberschreiben mit REAL INDEFINITE	
Ein oder beide Operanden sind NANs.	NAN ergibt sich als Operationsergebnis mit dem groessten Absolutwert (Vorzeichen bleiben unberuecksichtigt).	
Vergleichs-u.Testoperationen: Ein o.beide Operanden sind NANs.	CONDITION CODE setzen"nicht vergleichbar" C3 I C0 1 I 1	
Add.-Operationen: -Affines Modell f. REAL-Zahlenbereich ist gewaehlt, die Operanden sind oo mit entgegengesetztem Vorzeichen. oder -Projektives Modell f. REAL-Zahlenbereich ist gewaehlt, beide Operanden sind oo (Vorzeichen ohne Bedeutung).	REAL INDEFINITE ergibt sich als Operationsergebnis.	
Subtraktionsoperationen: -Affines Modell f. REAL-Zahlenbereich ist gewaehlt, die Operanden sind oo mit gleichem Vorzeichen.	REAL INDEFINITE ergibt sich als Operationsergebnis.	

*** ANLAGE 1 ***

Bedingung	maskierte Antwort	unmaskierte Antwort
u n g u e l t i g e O p e r a t i o n		
oder -Projektives Modell f. REAL-Zahlenbereich ist gewaehlt, beide Operanden sind oo (Vorzeichen ohne Bedeutung).		Interruptanforderung
Multiplikationsoperationen: oo * Ø oder Ø * oo	REAL INDEFINITE ergibt sich als Operationsergebnis.	
Divisionsoperationen: oo/oo o. Ø/Ø o. Ø/Pseudo-Null oder -Divisor ist ein DENORMAL -Divisor ist ein UNNORMAL	REAL INDEFINITE ergibt sich als Operationsergebnis.	
FPREM-Befehl: Divisor (Modulus) ist ein UNNORMAL oder DENORMAL oder Divident ist oo.	Operationsergebnis: REAL INDEFINITE und CONDITION-Bit C2 wird "Funktion beendet" gesetzt (C2=1)	
FSQRT-Befehl: -Operand ist nicht Null u. negativ oder -Operand ist ein DENORMAL bzw. UNNORMAL oder -das affine Modell fuer REAL-Zahlenbereich ist gewaehlt u.Operand ist -oo. oder -das projektive Modell fuer REAL-Zahlenbereich ist gewaehlt, der Operand ist oo.	Operationsergebnis: REAL INDEFINITE	

*** ANLAGE 1 ***

Bedingung	maskierte Antwort	unmaskierte Antwort
u n g u e l t i g e O p e r a t i o n		
Vergleichsoperationen: -Projektives Modell fuer REAL-Zahlenbereich ist gewaehlt, oo wird mit Ø verglichen oder mit einem NORMAL oder mit oo.	CONDITION-CODE "nicht vergleichbar" setzen C3 I C 1 I 1	Interruptanforderung
FTST-Befehl: -Projektives Modell fuer REAL-Zahlenbereich ist gewaehlt, Operand ist oo.	CONDITION-CODE "nicht vergleichbar" setzen C3 I C 1 I 1	
FIST-u.FISTP-Befehle: -Quellregister ist als EMPTY gekennzeichnet oder -im Quellregister steht NAN,DENORMAL,UNNORMAL o.oo oder -darstellbarer Bereich d.Zielformat wird ueberschritten.	INTEGER INDEFINITE speichern	
FBSTP-Befehl: -Quellregister ist als EMPTY gekennzeichnet oder -im Quellregister steht NAN,DENORMAL,UNNORMAL o.oo oder -eine Zahl, die 18 dezimale Stellen ueberschreitet	PACKED DECIMAL INDEFINITE speichern	
FST-und FSTP-Befehle: -Speicherziel ist	REAL INDEFINITE	

*** ANLAGE 1 ***

Bedingung	maskierte Antwort	unmaskierte Antwort
ungueltige Operation		
vom Typ SHORT o. o.LONG REAL, u.im Quellregister steht ein UNNORMAL, dessen Exponent im Bereich liegt.	speichern	Interruptanforderung
FXCH-Befehl: Ein oder beide Register sind als EMPTY gekennzeichnet.	Das als EMPTY gekennzeichnete Register wird mit REAL INDEFINITE geladen. Sind beide Register EMPTY,werden sie mit REAL INDEFINITE geladen. Danach werden die Registerinhalte vertauscht.	
denormalisierter Operand		
FLD-Befehl: Quelloperand ist ein DENORMAL.	Keine spezielle Aktion, wie ueblich Laden.	Interruptanforderung
Arithmetische Operationen: Ein oder beide Operanden sind DeNORMAL.	Operanden werden in aequivalente UNNORMALS konvertiert und dann die Operation ausgefuert.	
Vergleichs- und Testoperationen: Ein oder beide Operanden sind DENORMALS o.UNNORMALS (keine PSEUDO NULLEN).	DENORMALS werden in aequivalente UNNORMALS konvertiert und diese soweit wie moeglich normalisiert und dann die Operation ausgefuehrt.	
Division durch Null		
Divisionsoperationen: Divisor = 0	Quotient = oo, Vorzeichen d.Quotienten ergibt sich aus "Exklusiv-Oder", Verknuepfung d.Vorzeichen d.Operanden.	Interruptanforderung

Bedingung	maskierte Antwort	unmaskierte Antwort
	U e b e r l a u f	
Arithmetische Operationen: Rundungsarten: -Runden zur naechsten darstellb. Zahl oder -Abschneiden in Richtung Null, der Exponent des wahren Ergebnisses ist >16383.	Operationsergebnis oo vorzeichenbehaftet, Anzeige PRAECISION EXCEPTION.	Operationsergebnis im Zielregister, WM87 subtrahiert die Konstante 24576 D vom Exponenten des Ergebnisses u.bringt dadurch den Exponenten in den darstellbaren Bereich des TEMPORARY REAL-Formates. Danach erfolgt eine Interruptanforderung an die CPU. In der Interruptroutine (Exception Handler) kann die Konstante addiert werden, um den wahren Exponenten festzustellen. Das Operationsergebnis wird im Speicher (Ziel) hinterlegt. Interruptanforderung
FST, FSTP-Befehle: Rundungsarten: -Runden zur naechsten darstellb. Zahl oder -Abschneiden in Richtung Null, der Exponent des wahren Ergebnisses ist >+127 (SHORT REAL-Ziel) oder >+1023 (LONG REAL-Ziel).	Operationsergebnis: oo vorzeichenbehaftet, Anzeige PRECISION EXCEPTION	Interruptanforderung
	U n t e r l a u f	
Arithmetische Operationen: Der Exponent des wahren Ergebnisses ist <-16382.	Denormalisieren, bis der Exponent den wahren Wert -16382 erhaelt, Signifikand auf 64 Bits runden. Ist der denormalisierte gerundete Signifikand = 0,	Operationsergebnis im Zielregister: Der WM87 addiert die Konstante 24576 D zum Exponenten des Ergebnisses u.bringt dadurch den Exponenten in den darstell-

Bedingung	maskierte Antwort	unmaskierte Antwort
	U n t e r l a u f	
	dann "wahre "Null" speichern, anderenfalls ein DENORMAL.	baren Bereich des TEMPORARY REAL-Formates. Danach erfolgt eine Interruptanforderung an die CPU. In der Interruptroutine (Exception Handler) kann die Konstante subtrahiert werden, um den wahren Exponenten festustellen. Das Operationsergebnis wird im Speicher (Ziel) hinterlegt. Interruptanforderung
FST,FSTP-Befehle: Zielformat entspricht SHORT REAL Zahl, der Exponent des wahren Ergebnisses ist >-126.	Denormalisieren, bis Exponent den wahren Wert -126 erhaelt. Signifikand auf 24 Bits runden. Ist der denormalisierte gerundete Signifikand =0, dann "wahre" Null speichern, anderenfalls ein DENORMAL.	Interruptanforderung
FST,FSTP-Befehle: Zielformat entspricht LONG REAL-Zahl, der Exponent des wahren Ergebnisses ist <-1022.	Denormalisieren, bis Exponent den wahren Wert -1022 erhaelt. Signifikand auf 53 Bits runden. Ist der denormalisierte gerundete Signifikand =0, dann "wahre" Null speichern, anderenfalls ein DENORMAL.	Interruptanforderung
	P r a e z i s i o n	
Rundungsfehler, Operationsergebnis kann nicht im gewuenschten Ziel-	Das gerundete Ergebnis wird gespeichert.	Das gerundete Ergebnis wird gespeichert und dann eine Interruptanforderung aus-

Bedingung	maskierte Antwort	unmaskierte Antwort
	Praezision	
format dargestellt werden.		gegeben.
Beim Ausfuehren eines Befehles ist eine OVERFLOW EXCEPTION aufgetreten und eine maskierte Antwort gegeben worden.	keine spezielle Aktion	Interruptanforderung

Maskierte Antworten bei Overflow Exceptions abhaengig von der Rundungsart:

Wahres Ergebnis	Vorzeichen	Rundungsart	Ergebnis
NORMAL	+	up	+oo
NORMAL	+	down	groesste endliche positive Zahl *
NORMAL	−	up	groesste endliche negative Zahl *
NORMAL	−	down	−oo
UNNORMAL	+	up	+oo
UNNORMAL	+	down	Signifikand des wahren Ergebnisses und groesster Exponent **
UNNORMAL	−	up	Signifikand des wahren Ergebnisses und groesster Exponent **
UNNORMAL	−	down	−oo

* Die groesste darstellbare endliche REAL-Zahl ist kodiert:
Exponent 11...10B
Signifikand (1)^ 11...11B

** Der Signifikand behaelt seine Identitaet als UNNORMAL.
Das wahre Ergebnis wird wie gewoehnlich gerundet (in Richtung Null).
Der Exponent ist kodiert: 11...10B.

Definitionen / Abkuerzungen

CU Control Unit

NEU Numeric Execution Unit

Chop Abschneiden in Richtung Null

Bias Verschiebekonstante bei Exponenten von REAL-Zahlen

ST,ST(Ø) Spitze des Gleitpunktstack = Stacktop

ST(i),(i) 3-Bit-Feld, i-tes Stackelement nach Stacktop

mem-op Adresse eines WM87-Speicheroperanden

mode Bits 7 und 6 des MODRM-Bytes
Dieses 2-Bit-Feld definiert den Adressmode.

d 1-Bit-Zielindikator: Ø = Ziel Stacktop,
1 = Ziel Stackelement

r/m Bits 2, 1, Ø des MODRM-Bytes, bei Speicherzugriff-Operanden verwendet, dieses Feld definiert EA in Verbindung mit mode und m.

m 1-Bit-Feld, Angabe der Datentyplaenge,
REAL m = Ø SHORT REAL
m = 1 LONG REAL

INTEGER m = Ø SHORT INTEGER
m = 1 WORD INTEGER

| | Betrag, Absolutwert
θ Omega
oo Unendlich
+ Addition
- Subtraktion
* Multiplikation
/ Division
< Kleiner
> Groesser
<= Kleiner gleich
>= Groesser gleich
^ Im Signifikant-Feld z.B. 1^ØØ
: Vergleichen
Pi Konstante 3,14...
B Binaer
D Dezimal
H Hexadezimal

III-12-12 Kv 1454/88

NOTIZEN

NOTIZEN

NOTIZEN

NOTIZEN